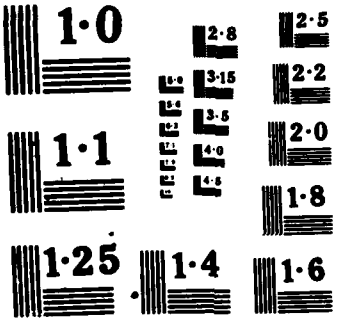
AD-A168 298 GENERALIZED NETWORK IMPLEMENTATIONS(U) GEORGIA INST OF TECH ATLANTA PRODUCTION AND DISTRIBUTION RESEARCH CENTER J J JARVIS ET AL. 1986 PDRC-86-03 1/2

UNCLASSIFIED N00014-85-C-0797 F/G 5/1 NL



Generalized Network Implementations

John J. Jarvis
H. Donald Ratliff
Michael A. Trick

PDRC 86-03

Report for:
Joint Deployment Agency
MacDill Air Force Base, FL 33608

Generalized Network Implementations

John J. Jarvis
H. Donald Ratliff
Michael A. Trick

PDRC 86-03

Report by:
School of Industrial and Systems Engineering
Georgia Institute of Technology
Atlanta, GA 30332

This work is supported by the Office of Naval Research under Contract No. N00014-85-C-0797 and N00014-83-K-0147. Reproduction in whole or in part is permitted for any purpose of the U. S. Government

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

Generalized networks are an important class of optimization models, with uses in a wide variety of fields. This report describes the development and implementation of a generalized network algorithm.

In [3], Jarvis et. al. recommend a generalized network model, system for closure optimization and planning (SCOPE), for crisis action deployment planning. In SCOPE, large generalized networks must be repeatedly solved. These networks have special structure, which results in computational advantages.

In this report, a generalized network implementation is developed for solving very large generalized networks. This implementation includes new data structures for storing the basis, in-core/out-of-core handling of the arcs, and special handling of pure network structure.

In this report, a detailed examination of the SCOPE model is provided and its effect on implementation issues is d scussed. The SCOPE model is highly structured. This report demonstrates how this structure can be used to advantage. In a companion report [4], extensive testing is presented which addresses the question "What affects the computation time for a SCOPE model?"

Section 2 provides an introduction to modeling with pure and generalized networks. Section 3 gives an overview of linear programming, as it applies to generalized network solution methods. In section 4, the special structure of a generalized network basis is detailed and efficient storage methods are developed. Section 5 gives methods for handling the arc data. This includes both storage

→ Keywords: linear programming; data processing; data structure systems. ←

methods and techniques for determining the arc to enter the basis. A method for determining the arc to leave the basis is given in section 6. Section 7 describes, in detail, the algorithms used to update the basis structures. Section 8 discusses some related implementation concerns. These include the initial basis to be used, and the effect of embedded pure network structure. Section 9 gives a review of the conclusions.

2.0 MODELING WITH GENERALIZED NETWORKS

Generalized networks, as the name suggests, are a generalization of standard, or "pure", networks. By modifying a restriction that occurs in pure networks, many previously intractable problems have been modeled. Section 2.1 provides a brief overview of pure network modeling. Section 2.2 expands this overview to include generalized networks. Section 2.3 details the SCOPE generalized network model. The SCOPE model is an example of generalized network model. It will form the basis for the examples used later in this report.

2.1 Pure Networks

A "pure" network can be thought of as a pipeline system. This system has suppliers and users of the materiel flowing through the pipe. Each of the suppliers and users has a known supply and demand respectively. The pipeline connects the suppliers and users. The pipeline may have intermediate junction points which are not suppliers or users. (An example would be a pumping station.)

Each pipe has a known capacity. This capacity represents the limit on the pipe expressed in the rate of flow of materiel through the pipe. There is a unit cost associated with materiel that flows through the pipe. This cost is linear in the quantity of materiel; in other words, if the amount flowing through a pipe is doubled, then the cost is also doubled.

The objective is to move the materiel from the suppliers to the

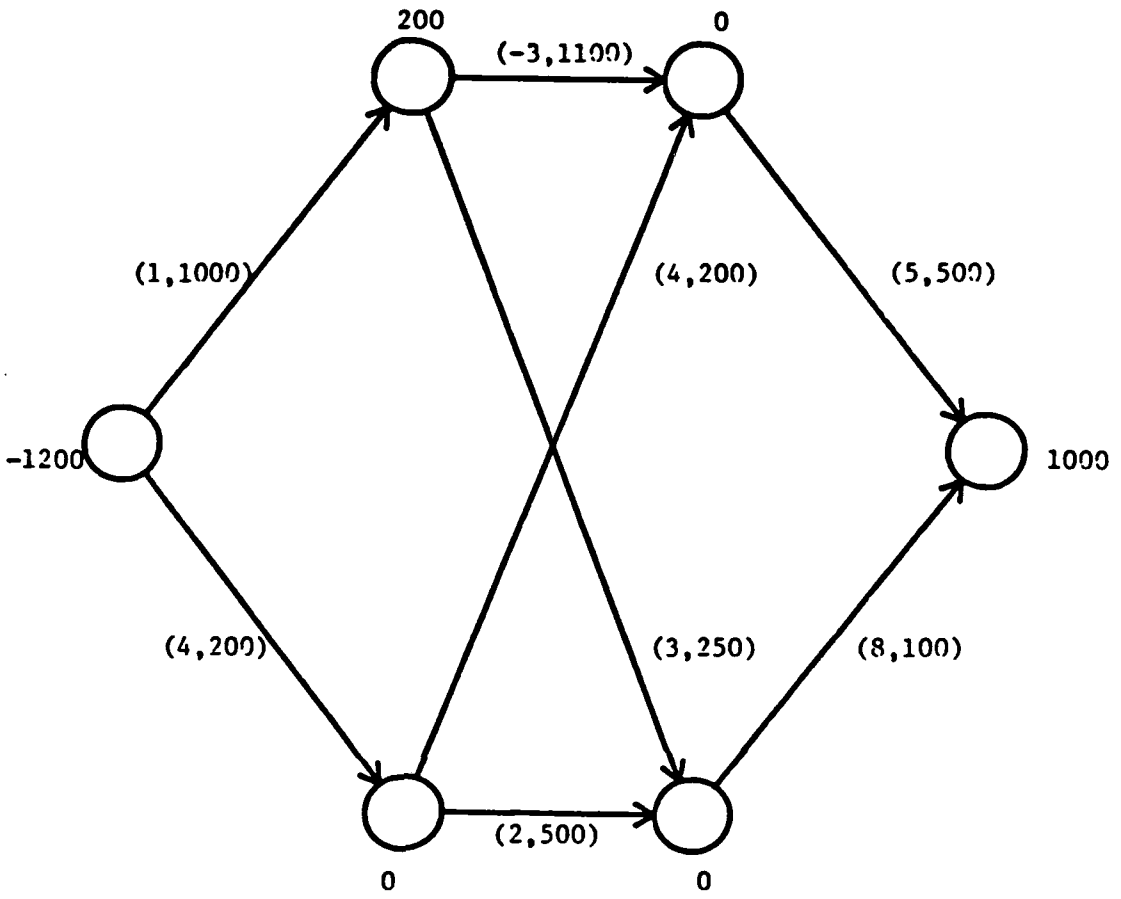
users through the pipeline at minimum cost. This involves assigning flows to pipes so that (1) each user gets the amount needed; (2) no supplier sends more than available; (3) no pipe has more materiel than its capacity; (4) no materiel enters or leaves the system except at users or suppliers; and (5) what enters the pipe at one end, leaves it at the other.

The suppliers, users, and junctions where two or more pipes come together may be represented by points, called nodes. The lengths of pipes between points (nodes) are called arcs. Associated with each node is a number, called its requirement. If the requirement is negative, then the node is a supplier, and the number is the amount that it can supply. If the requirement is positive, then the node is a user, and the number represents the amount it demands. A zero requirement can be used for junctions where arcs meet without representing a supplier or user.

Associated with each node is a flag. The flag indicates whether the requirement must be met exactly, or whether the absolute value of the requirement represents an upper bound for the supplier (or user). This accommodates models with nodes in which suppliers must ship the full amount of their supply and for users that may or may not use the full demand indicated for them.

Arcs have capacities, cost and direction. The node that an arc begins at is called its "TAIL". The ending node is the "HEAD". For arc number ARC these two ends are referred to as TAIL(ARC) and HEAD(ARC) respectively.

Figure 2-1 shows a sample network. The circles represent nodes. The lines (arrows) between them are the arcs.



label on arcs: (cost,capacity)
label on nodes: demand

Figure 2-1. Network Representation

2.2 Generalized Networks

Generalized networks are extensions of pure networks. The difference is that the restriction that "flow into an arc equals flow out of the arc" is relaxed. Instead, a generalized network allows for "leaky" arcs or arcs that gain flow. The loss or gain is specific to each arc and can vary throughout the network. The only requirement is that the arc must gain or lose a constant fraction of its flow. For instance, an arc might always triple its flow; another pipe might always quarter its flow. The fraction that the arc changes the flow is called the multiplier. The multipliers for the example arcs are 3 and 0.25 respectively. As a matter of convention, costs are calculated by multiplying the flow that enters the arc by the arc cost.

Because of gains and losses in a generalized network, it is impossible to require that supply equals demand, as is normal in pure network models. "Slack" and "surplus" arcs are required to model the excess supply or demand that normally occurs in a generalized networks. These "slack" and "surplus" arcs are modeled using "self-loops" at each node. "Self-loops" are arcs that begin and end at the same node. Surplus arcs, associated with demand nodes, have multipliers of +1. Slack arcs, attached to supply nodes have multipliers of -1.

All other properties of pure networks hold for generalized networks. A generalized network with all multipliers equal to one is a pure network. An arc with a multiplier of one is called a pure arc.

Generalized network models are very useful in many problems.

1) In a financial model, materiel flowing in a network represents money and nodes represent various points in time. Multipliers can be used to represent increase in money due to interest.

2) In an energy allocation model, materiel flowing in a network represents electricity, and arcs represent physical wires. Multipliers can be used to model energy losses that result when electricity flows along a wire.

3) In a deployment model, materiel flowing in the network are men and equipment to be moved. Arcs represent movement, by either airplane or ship. If materiel is moved by air then the weight (STONS) of the materiel determines the amount that can be moved. If movement is by sea, then the volume (MTONS) or square footage of the movement requirement is critical. Multipliers can be used to model conversion of weight into volume.

2.3 The SCOPE MRMATE Model

In [3], a method for solving a large deployment problem was presented. This method, called System for Closure Planning and Evaluation (SCOPE), addressed the following problem:

Given a set of assets (airplanes and ships), a set of movement requirements (people, ammunition, etc.), and a set of ports to use, is there a way to move the movement requirements with the available assets through ports so that the requirements arrive at the target area when needed?

A much more detailed examination of the problem is provided in [3]. The method proposed is based on three optimization components: a network flow with side constraints which assigns assets to pairs of

ports; a generalized network flow which assigns the movement requirements to assets; and a Benders' constraint generator to link together the two models. This report is concerned only with the second of the optimization pieces.

Fundamental to the SCOPE method is the concept of channels. A channel consists of a pair of ports, together with a number of identical assets. One port, the Port of Embarkation (POE), is where the movement requirements will be loaded onto the assets. The other port, the Port of Debarkation (POD), is the destination of the assets. The assets are assumed to cycle between the POE and POD. Depending on the distance between the POE and POD, the assets may be able to make one or more trips between the POE and POD in a single time period (which can be taken as a day for simplicity). Or, if the ports are far apart or the asset moves slowly, it may take several days to cycle between the POE and the POD. The capability of the channel is the rate at which the assets deliver movement requirements to the POD from the POE.

The first optimization model in SCOPE determines the channel capabilities. The second, generalized network model, must assign the movement requirements to channels at specific time periods so as to meet strategic objectives.

Suppliers in the SCOPE model are the movement requirements. Users are time expanded channels. Time expansion refers to the fact that a channel can move a given amount on each day. Nodes will be created for the channel for each day.

Certain movement requirements cannot be moved on specific channels. For instance, a movement requirement may not be air transportable, so they cannot use channels using air assets. There

are many other restrictions. There is a certain delay involved in getting a movement requirement to the POE, so even if a movement requirement can use a channel, it may not be able to use it on certain days. Arcs are created from each movement requirement to each time expanded channel node that the movement requirement can use. (See Figure 2-2).

The interpretation of flow on an arc from movement requirement M to channel C on day D is that amount of M will arrive at the POE associated with C on day D and will use the assets associated with C to be transported to the POD of C. From this information, it is possible to determine when M will arrive at its final destination. The cost associated with an arc depends on the value of getting M to its final destination at that time.

There are many ways of assigning costs to the arcs, but one of the most flexible involves time windows. Each movement requirement has a window of days in which it is desired to arrive at its final destination. If it arrives at its final destination within its window there is no cost. If it arrives outside its window (either early or late) then the cost is a function of how many days early or late it arrives.

The final complication is in how to measure the size of a flow. There are two types of channels: air and sea. The major limitation on the amount an air channel can move is weight (STONS) of the movement requirements. The major limitation on the sea channels is volume (MTONS). Each movement requirement has a weight and volume, and the relationship between these two values depends on the movement requirement.

To model this, the weight of the movement requirement is taken as

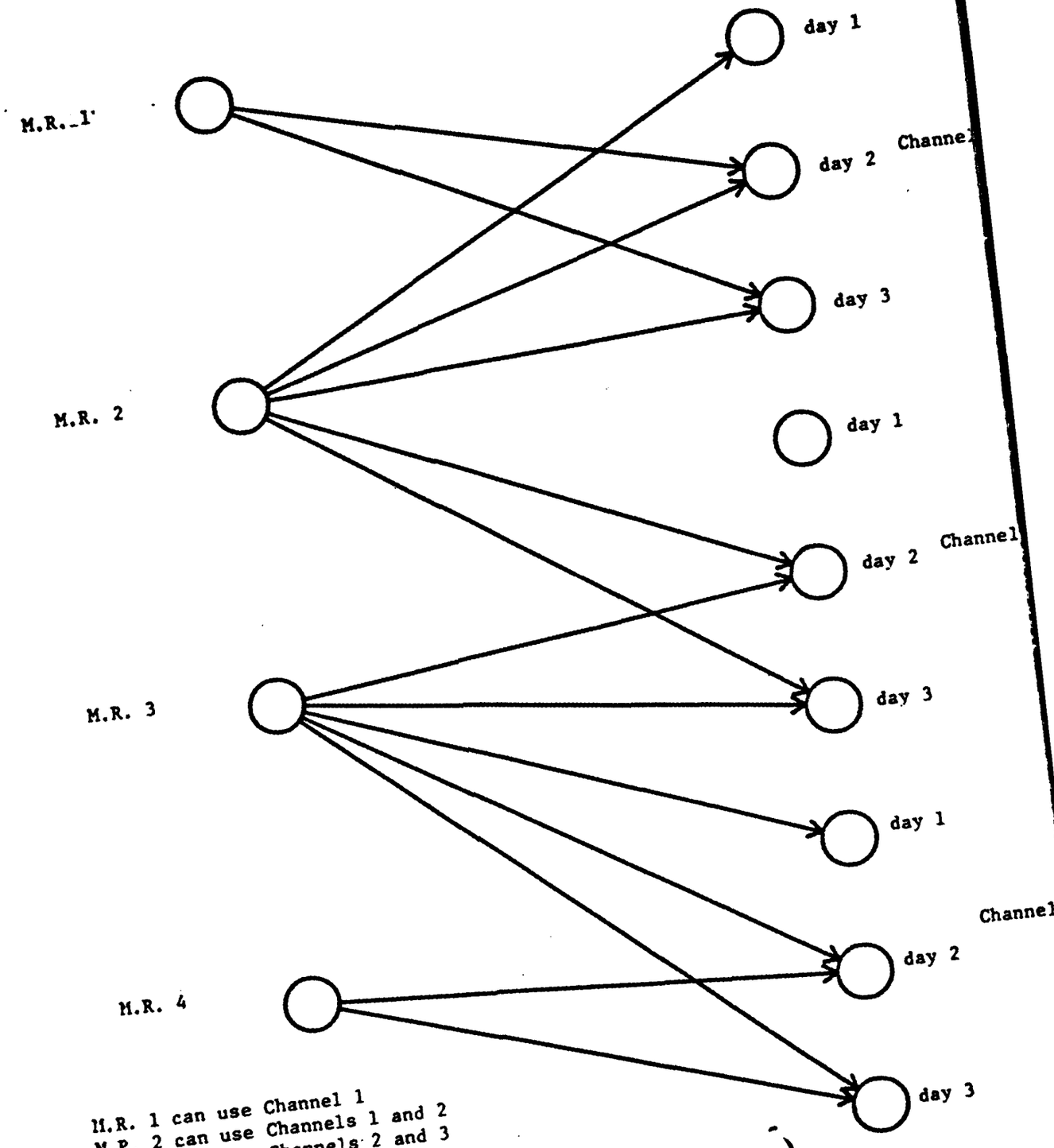


Figure 2-2. SCOPE Basic Model

its size. If the movement requirement is sent by an air channel, no conversion takes place. If the movement requirement is sent by a sea channel, the weight is converted to volume by a multiplier on the arc. For instance, if a movement requirement veighs 20 STONS and has volume 50 MTONS, arcs to air channels will be pure arcs (have multiplier one) and arcs to sea channels will have multipliers of 2.5 (=50/20). Figure 2-3 provides the complete network for the example.

The network created, called the MRMATE network, has significant structure. The major features are:

1) all nodes are either suppliers or users. In fact, this network is a transportation network (see Bazaraa and Jarvis [1]).

2) many of the multipliers are one.

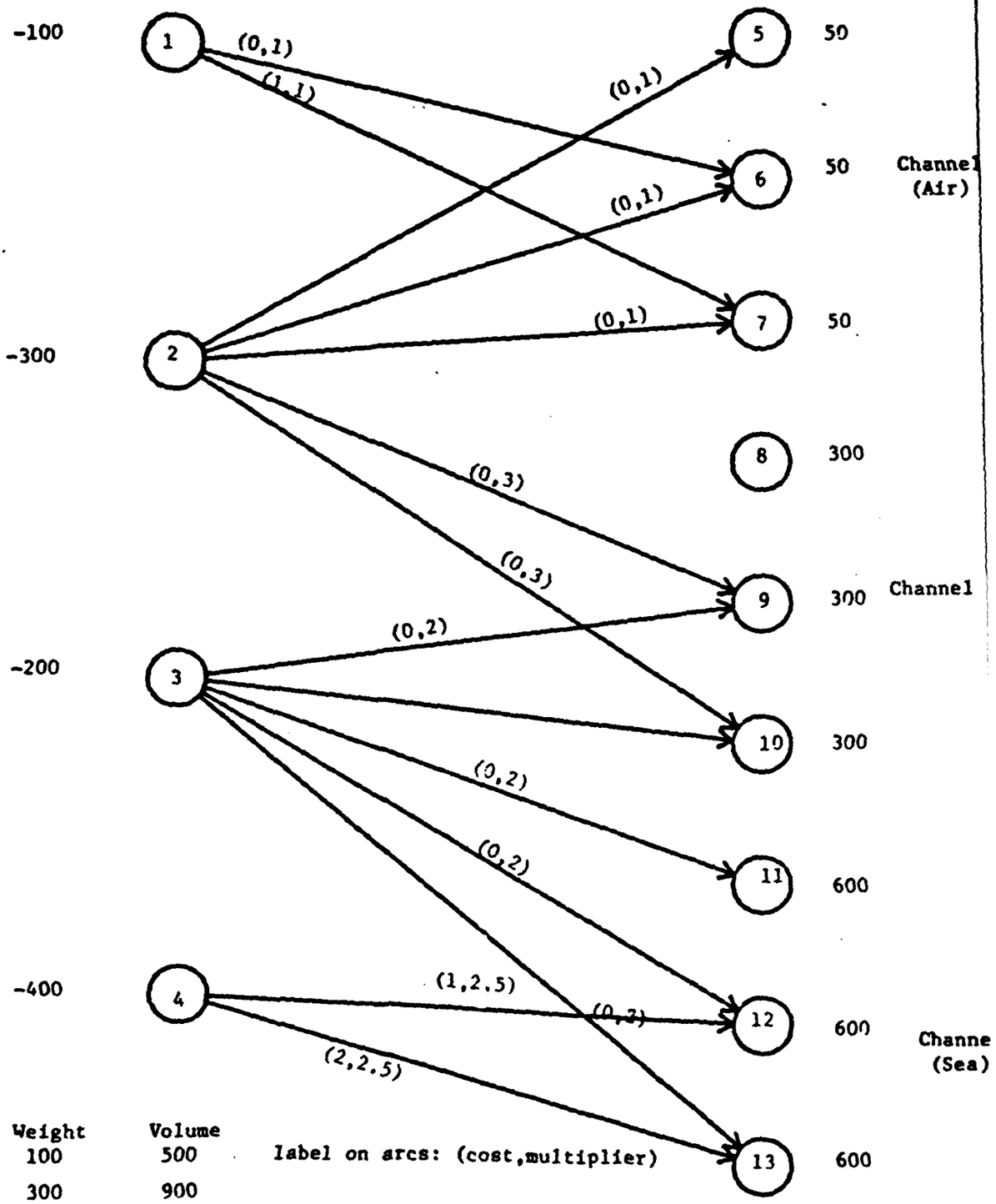
3) all arcs out of a movement requirement have one of two multipliers.

3) many arcs have zero cost.

4) the arcs have no capacities.

The largest problem that could occur in practice is estimated to have approximately 2000 nodes and over 500,000 arcs. Furthermore, these problems must be solved repeatedly in a very short amount of time.

The MRMATE model will be used as an example of the type of specializations possible in implementations of generalized networks in the remainder of this report.



	Weight	Volume
MR1:	100	500
MR2:	300	900
MR3:	200	400
MR4:	400	1000

label on arcs: (cost,multiplier)

Figure 2.3. SCOPE Generalized Network Model

3.0 SOLVING GENERALIZED NETWORKS

Since the generalized network model is so useful, it is important to have a computer solution technique that will find solutions quickly. It can be shown that generalized networks are a special case of linear programming, so that any technique to solve linear programming, like the Simplex method, can be used to find solutions to this model. But there are disadvantages of using these general purpose algorithms. The best known methods take too much space and are relatively slow. Fortunately, the Simplex method can be specialized so as to take advantage of the special structure in a generalized network. The specialized simplex method solves generalized networks quickly using little space.

Section 3.1 Linear Programming

Linear programming is the most fundamental optimization model in operations research. Bazaraa and Jarvis [1] provide an excellent introduction to this field. The linear programming model employs the optimization of a linear function subject to a set of linear constraints. A linear function is a function that is of the form:

$C_1 x_1 + C_2 x_2 + ... + C_n x_n$

where each of C_1, C_2, ... C_n are constants

and x_1, x_2, and x_n are the variables.

A linear constraint is of the form:

$A_1 x_1 + A_2 x_2 + ... + A_n x_n = B$

where A_1, A_2, ... A_n and B are all constants.

When a generalized network is represented as a linear program,

variable x_i is associated with each arc. This variable represents the amount of flow in the arc.

There is a linear constraint for each node. This constraint controls the amount of the flow that exits or enters the node. The B value for the constraint is the supply or demand for that node. The constraint forces the net amount of flow at a node (including the self-loop), to be the supply or demand for the node.

Every arc is associated with just two nodes: the tail and head nodes for the arc. This implies that the variable associated with each arc occurs in at most two constraints in the linear program. Self-loops occur in only one constraint.

In matrix terms, each constraint represents a row of the constraint matrix and each variable represents a column. The preceding argument indicates that there are at most two non-zero elements in each column of the constraint matrix.

Because the definition of generalized networks in this report allows just a single multiplier for each arc, one of the non-zeros of each column will be the multiplier on the arc. For arcs that are not self-loops, the other non-zero element of the column will be -1. The multiplier will be in the row associated with the constraint on the head of the arc. The -1, for non-self-loops, is associated with the tail of the arc.

Every variable is assumed to be constrained to be nonnegative. There are standard "tricks" to transform variables not of this form to the assumed form.

Since the arcs (variables) have capacities, it is necessary to treat this problem as a linear program with upper bounds. These upper bounds are linear constraints themselves. Due to the simplicity

of upper bounds it is possible to treat them implicitly in the solution algorithm.

3.2 Solving Linear Programs - The Primal Simplex Method

There are many methods for solving linear programs. The most widely used is the primal simplex method. This technique has proved to be efficient, both in execution time and computer space.

For every linear program, there is an optimal solution with no more than one non-zero variable for each constraint. This is referred to as a basic optimal solution. The optimal non-zero variables form a basis. A basis is any set of variables with the following properties:

1) There are not more variables than constraints in the linear program.

2) No column of the constraint matrix for any variable in the basis can be written as a weighted combination of the columns of the other variables in the basis.

3) There is a feasible solution to the linear program using just the variables in the basis.

The steps of the primal simplex method are as follows:

1) Find an initial basis.

2) Find a variable, not in the basis, to enter the basis. If none exists, STOP. The current basis is optimal.

3) Find the variable in the basis that will leave.

4) Update the basis

5) GOTO Step 2.

One iteration of steps 2 through 4 is called a "pivot".

Step 1 can be accomplished in various ways. The simplest method is to take the "slack" and "surplus" variables that often occur in a linear program and use them as the initial basis. Sometimes artificial variables must be added where "real" slack and surplus variables do not exist. These artificial variables are given a high cost, so that the optimal solution will not employ any of them.

It is often possible to determine a set of variables that creates a very good solution. This usually reduces the number of pivots required to reach optimality. The time to find a good starting solution, called an advanced start, must be short enough not to offset the reduced computation time for the rest of the algorithm.

Identifying a variable to enter the basis is accomplished by determining the change in objective function if the variable is increased by a small value. This change is called the reduced cost.

Increasing the value of the variable entering the basis will change the values of the current basic variables. One of these variables will be the first to reach zero. This is the variable to exit the basis.

The new basis consists of the old basis, without the exiting variable, and the entering variable. Various values must be updated, including the new variable values and the reduced costs for variables not in the basis.

The primal simplex method can be adapted for upper bounds on variables. Rather than treat the upper bounds as "normal" constraints, which would be inefficient, the definition of basis is slightly redefined. A non-basic variable can now have value of either zero or its upper bound. A basic variable can have any value between zero and its upper bound. A non-basic variable at its upper

bound may enter the basis if decreasing its flow slightly improves the objective function. When the basic variables change value (Step 3) one of them will reach its upper bound or zero first. That variable will be the exiting variable.

4.0 STORING THE BASIS

The main reason that a specialized simplex method is faster than a general purpose simplex method for generalized networks is that the basis has a special structure. This structure makes every simplex computation easier. This section defines the basis structure and gives data structures to efficiently store it.

4.1 Basis Definition

A basis in linear programming consists of a set of columns, one for each row, with the property that no column is a weighted sum of the others. In a generalized network, columns correspond to arcs, so the basis is a set of arcs. Rows correspond to nodes, so there is one arc in the basis for each node. The final property, called linear independence, is more complicated to describe.

If a set of arcs is examined, the set of nodes will be partitioned into sets of nodes that are connected to each other (see Figure 4.1). These sets are called components. Within a component, the arcs can form cycles. Self-loops are treated as cycles of length 1. A component can have zero, one, or more than one cycle (components A, B, and C respectively in Figure 4.1). It is possible to show that if a component has more than one cycle, there is at least one arc that is the weighted sum of the other arcs in the component. Therefore, for a set of arcs to form a basis, it is necessary that no component formed by the arcs have more than one cycle. It is also possible to show that if a component has no cycle, some other component must have more than one cycle. Therefore, it is

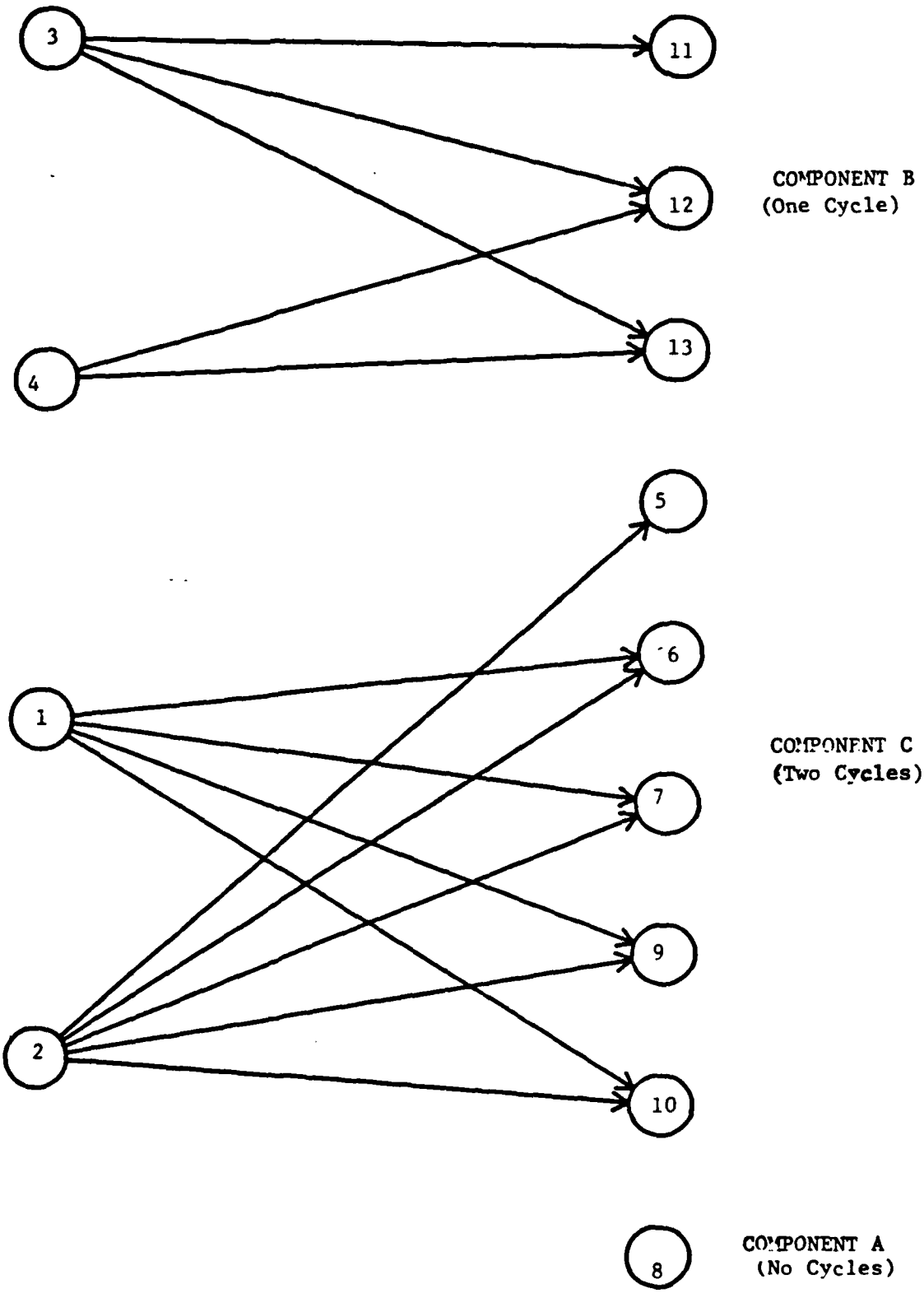


Figure 4-1. Components

also necessary that every component have at least one cycle. So every component has exactly one cycle. A component with exactly one cycle is called a "one-tree".

One further condition is required to ensure linear independence. If the cycle is not a self-loop, it is possible for an arc in the cycle to be a weighted sum of the other arcs in the cycle. A necessary and sufficient condition for this not to occur is for the cycle to have a cycle multiplier not equal to one. The cycle multiplier is calculated as follows: Assign an orientation to the cycle (clockwise or counter-clockwise). The cycle multiplier is the product of the arc multipliers for those arcs pointed in the same direction as the orientation, divided by the product of the arc multipliers of those arcs pointed in the reverse direction as the orientation.

To summarize, a set of arcs is a basis if the following conditions are satisfied:

1) The number of arcs is equal to the number of nodes.

2) Each component has exactly one cycle.

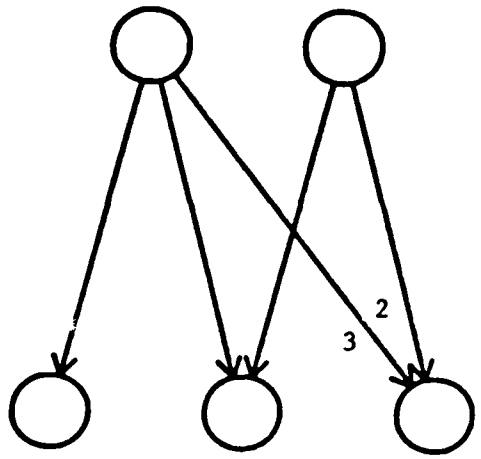
3) Each component with a cycle that is not a self-loop has a cycle multiplier that is not equal to one.

Some valid and invalid basis examples are given in Figure 4-2.

Since every multiplier in a pure network is 1, it is not possible for a pure network basis to have a cycle that is not a self loop.

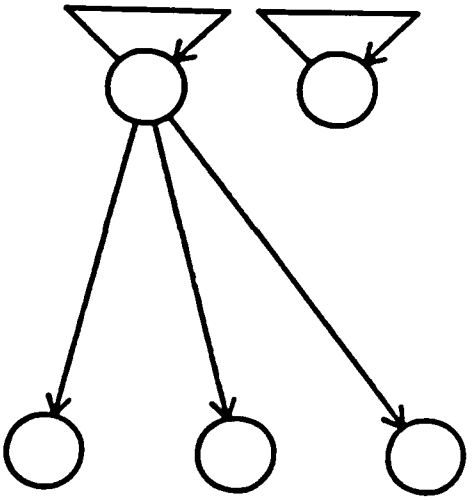
4.1.1 Specialization for the SCOPE Model

The arc multipliers in the MRMATE model have a special structure. Since every arc connects a movement requirement to either an air or a

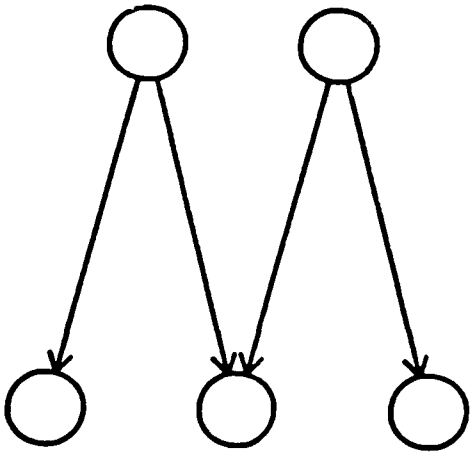


(a) valid

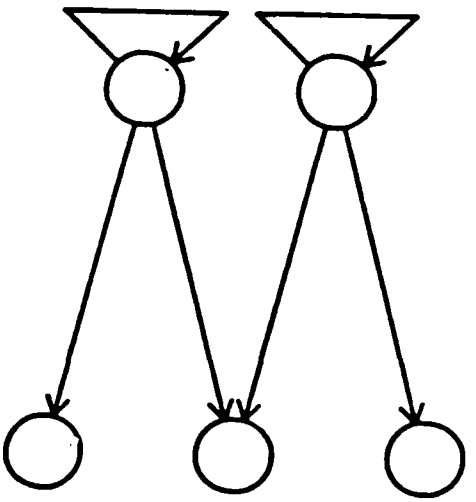
label: multiplier



(b) valid

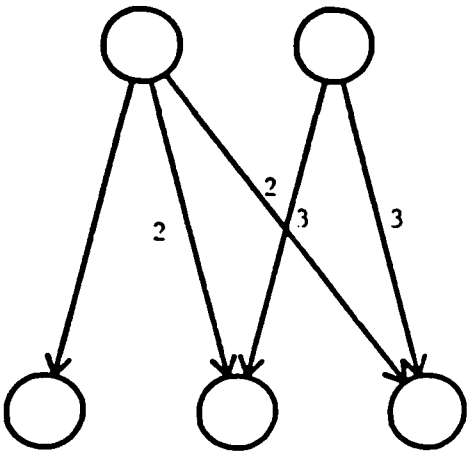


(c) invalid (no cycle)



(d) invalid (two cycles)

Figure 4-2. Valid and Invalid Basis



label on arc: multiplier

(e) invalid: cycle multiplier of 1

Figure 4-2. Continued

sea channel, the arcs have multipliers of either 1 or the conversion factor for the movement requirement. Given the way that cycles form in the MRMATE model, it is easy to show that if a component has just air channel arcs or just sea channel arcs, then the cycle associated with the component must be a self loop. In other words, if a component has only one type of arc then that component has the same basis structure as a pure network basis. Since pure networks can be solved more efficiently than generalized networks, it is likely that some advantage can be taken of the basis structure in this case.

4.2 Storing the Basis

Since the basis for generalized networks has special structure, it should be possible to store the basis in an efficient way. There are two important factors in storing the basis: storage space and computation time.

The following sections outline a method of storing the basis, called the linked rooted tree method. This method is similar to that of Brown and McBride [2], but differs in some important ways.

The linked rooted tree method is based on the data structures used for pure networks (see Kennington [5]). In this method, nodes on the cycle are seen as roots for trees consisting of nodes not on the cycle. These trees are then linked around the cycle (see Figure 4-3). Each component contains one or more trees together with the linking cycle.

4.2.1 Predecessor Structure

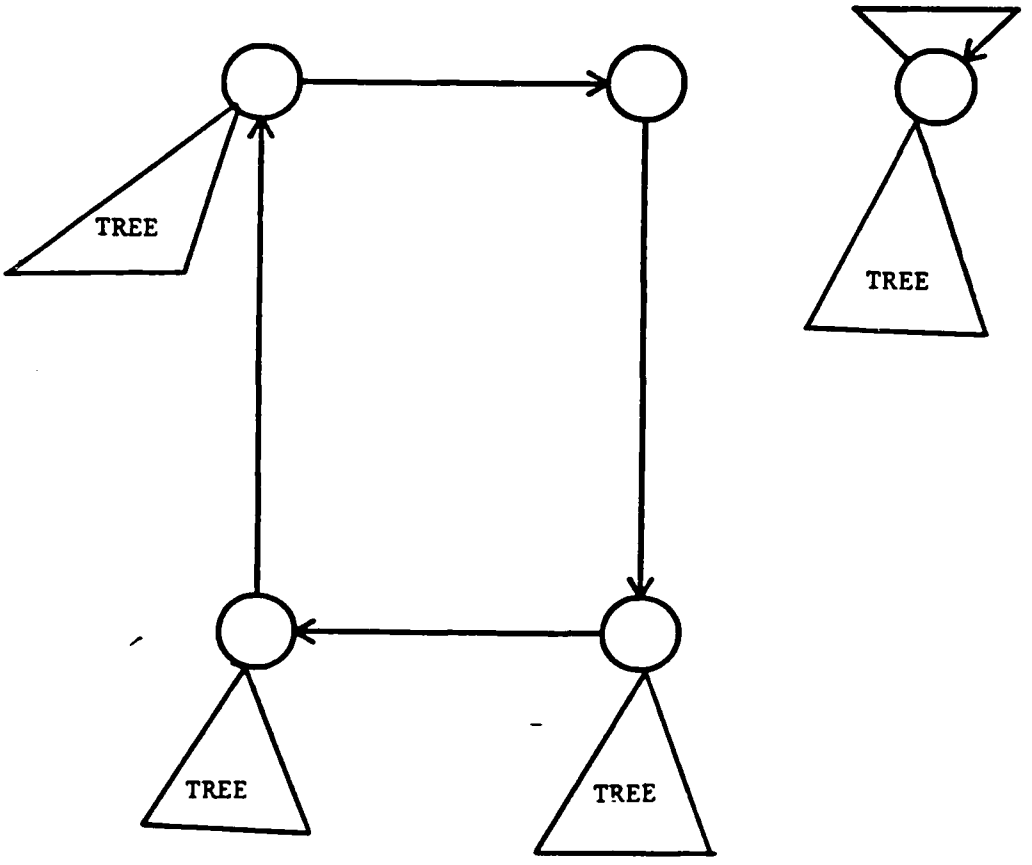


Figure 4-3. Linked Rooted Trees

The most fundamental operation required for the manipulation is to "go up" the tree. (Here the cycle is considered "on top" of the tree). This is required in determining the arc to exit the basis, for the arcs that must be checked are exactly those above the endpoints of the entering arc. It is also very useful when determining the new basis (Section 7).

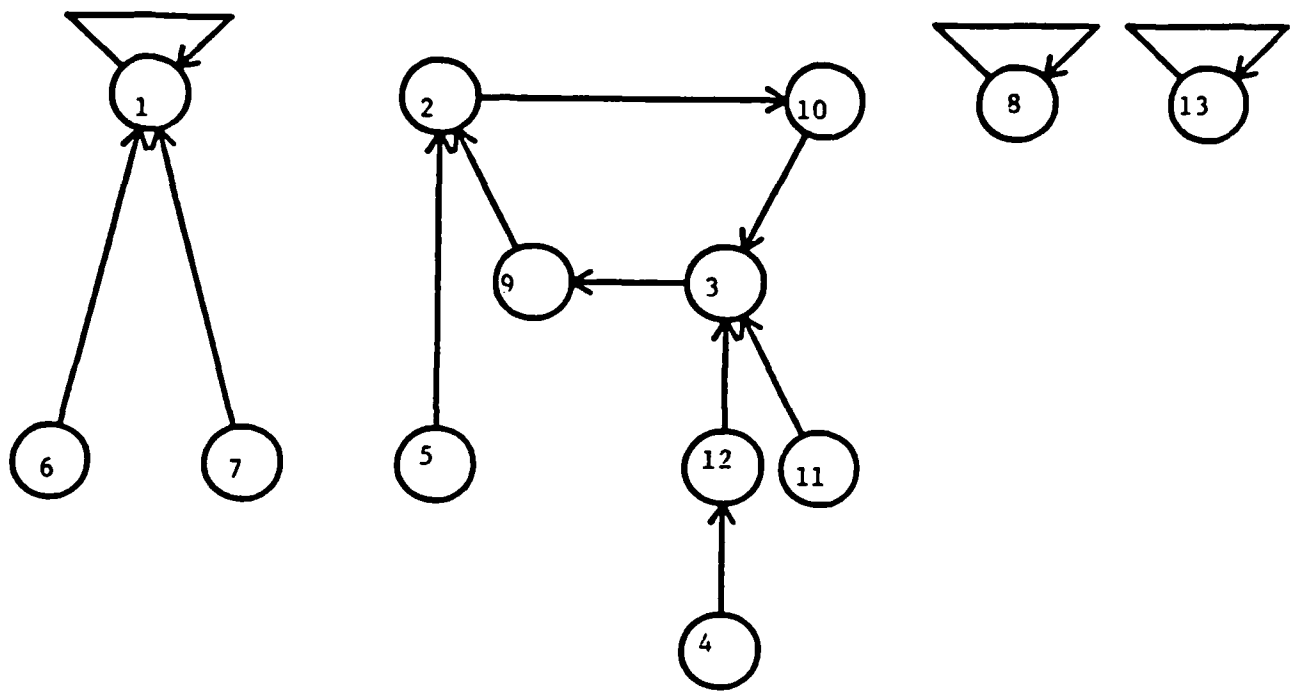
For any selected node not on the cycle, the predecessor (PRED) is defined to be the (unique) node, such that there is a basic arc connecting the two nodes and PRED is closer to the cycle than the selected node.

For nodes on the cycle, an arbitrary orientation of the cycle is selected. The PRED of a node on the cycle is the node just before it on the cycle using the selected orientation. The PRED of a node on the cycle is also on the cycle. If the cycle is a self-loop, the PRED of the cycle node is defined to be itself. (See Figure 4-4).

The PRED provides the only method of moving from one cycle node to another under the linked rooted tree system. Also note that there is no connections between components, for that ability is not required for the simplex calculations.

4.2.2 Thread Structure

The thread structure (THREAD) provides a mechanism for visiting every node in a tree. The order in which the nodes are visited is defined to be the "preorder traversal" (see Kennington [5]). This order has the property that if node X is on the path from Y to the root then node X is visited before node Y. Note that the THREAD is only within trees, not between them. (See Figure 4-5).



Node	Pred
1	1
2	10
3	9
4	12
5	2
6	1
7	1
8	8
9	2
10	3
11	3
12	3
13	13

Figure 4-4. Predecessor Structure

The THREAD is required by the basis update routines to determine those nodes whose duals and LEVELs (Section 4.2.3) must be updated.

It is in this structure that the linked rooted tree method differs from that used by Brown and McBride. In that report, the THREAD was defined traverse around the cycle in the opposite direction of PRED.

4.2.3 Level Structure

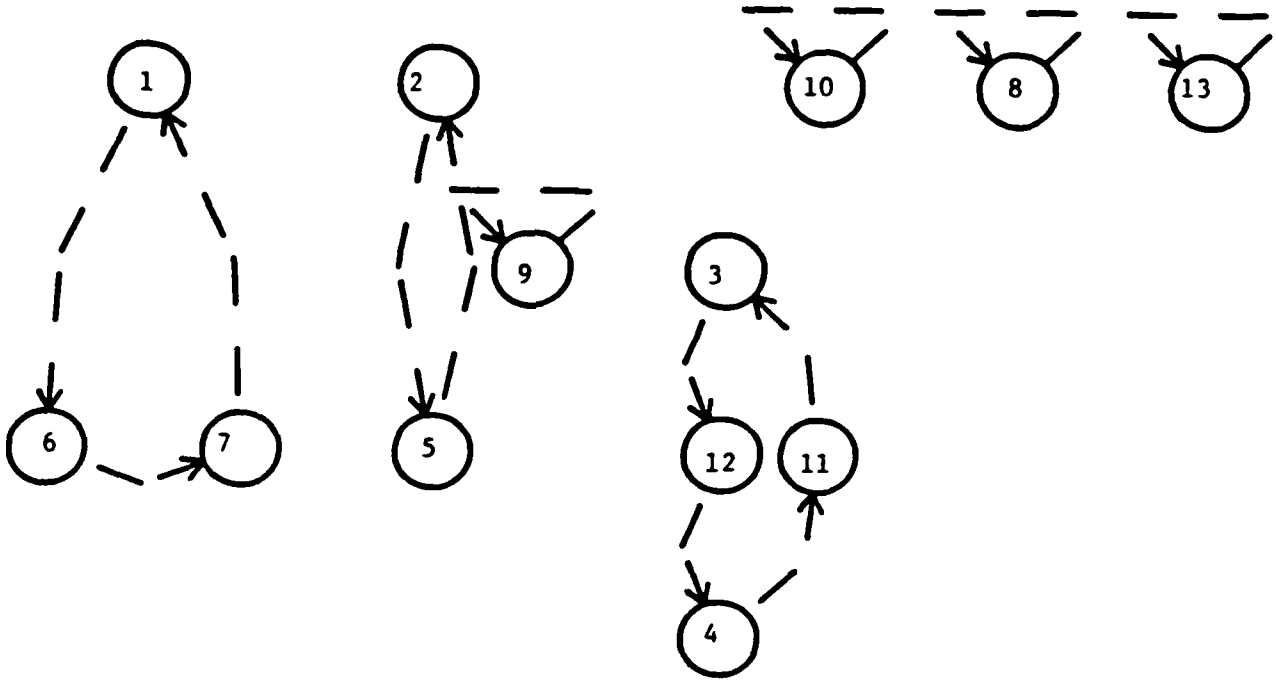
The level structure (LEVEL) gives the "distance" of a node to the cycle. Nodes on the cycle have a LEVEL value of zero. This structure is required by the routine to find the exiting arc (Section 6). (See Figure 4-6).

4.2.4 Reverse Thread Structure

The reverse thread structure (RTHREAD) is simply the inverse of the THREAD structure. This permits the visiting of nodes in reverse order. Typically, this structure is only used to make the basis update more efficient. (See Figure 4-7)

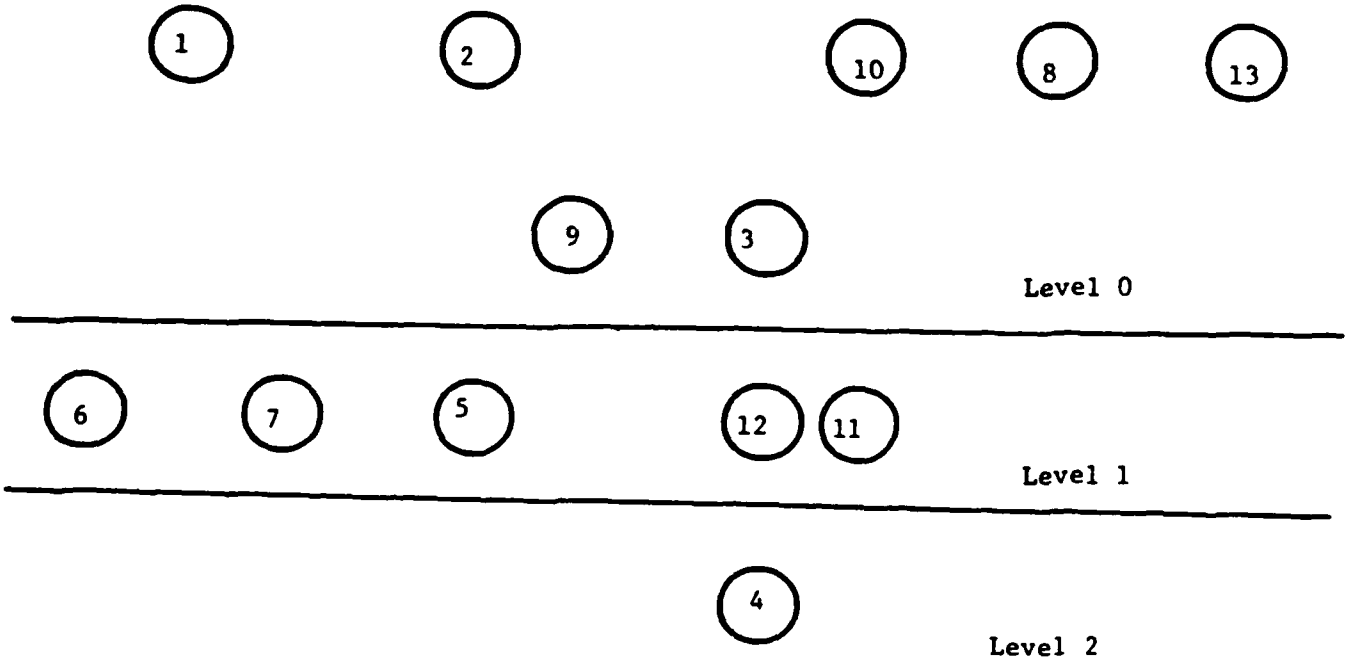
4.2.5 Arc Information Structures

Some information on the basic arcs is required to perform the simplex calculations. This includes the CAPACITY, arc multiplier (MULT) and current FLOW, to determine the arc to exit the basis; the arc COST, to update the dual variables; and the arc number (ARC) to record the optimal solution. If the information on all arcs (basic



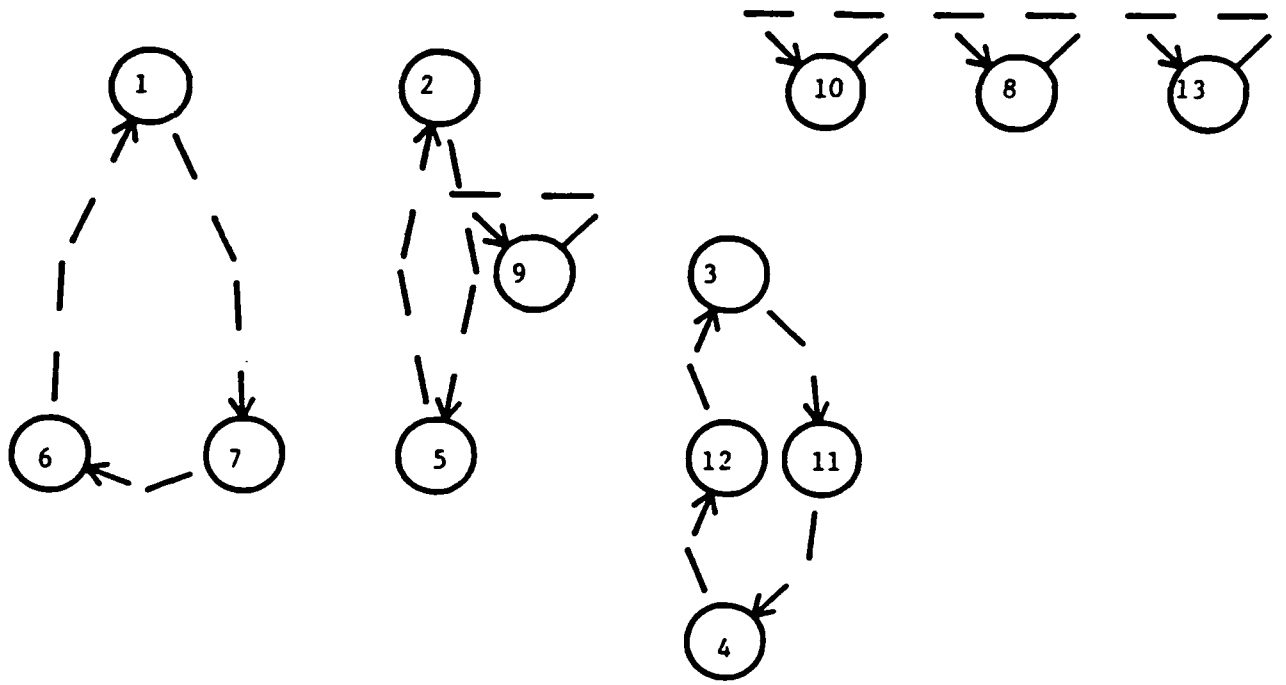
Node	Thread
1	6
2	5
3	12
4	11
5	2
6	7
7	1
8	8
9	9
10	10
11	3
12	4
13	13

Figure 4-5. Thread Structure



Node	Level
1	0
2	0
3	0
4	2
5	1
6	1
7	1
8	0
9	0
10	0
11	1
12	1
13	0

Figure 4-6. Level Structure



Node	Reverse Thread
1	7
2	5
3	11
4	12
5	2
6	1
7	6
8	8
9	9
10	10
11	4
12	3
13	13

Figure 4-7. Reverse Thread Structure

and non-basic) is available, then it is only necessary to store the ARC value explicitly. As Section 5.1 will show, however, for large problems it is necessary to have only a limited amount of arc information available at any given time. Therefore, all of the above basic arc information must be stored.

The information on the basic arc that connects NODE and PRED(NODE) is associated with NODE. Since it is not clear whether the arc begins at NODE and ends at PRED(NODE) or the reverse, the ARC value is given a sign depending on the orientation. In the former case, ARC is positive; in the latter, ARC is negative. (See Figure 4-8).

4.2.6 Dual Value Structure

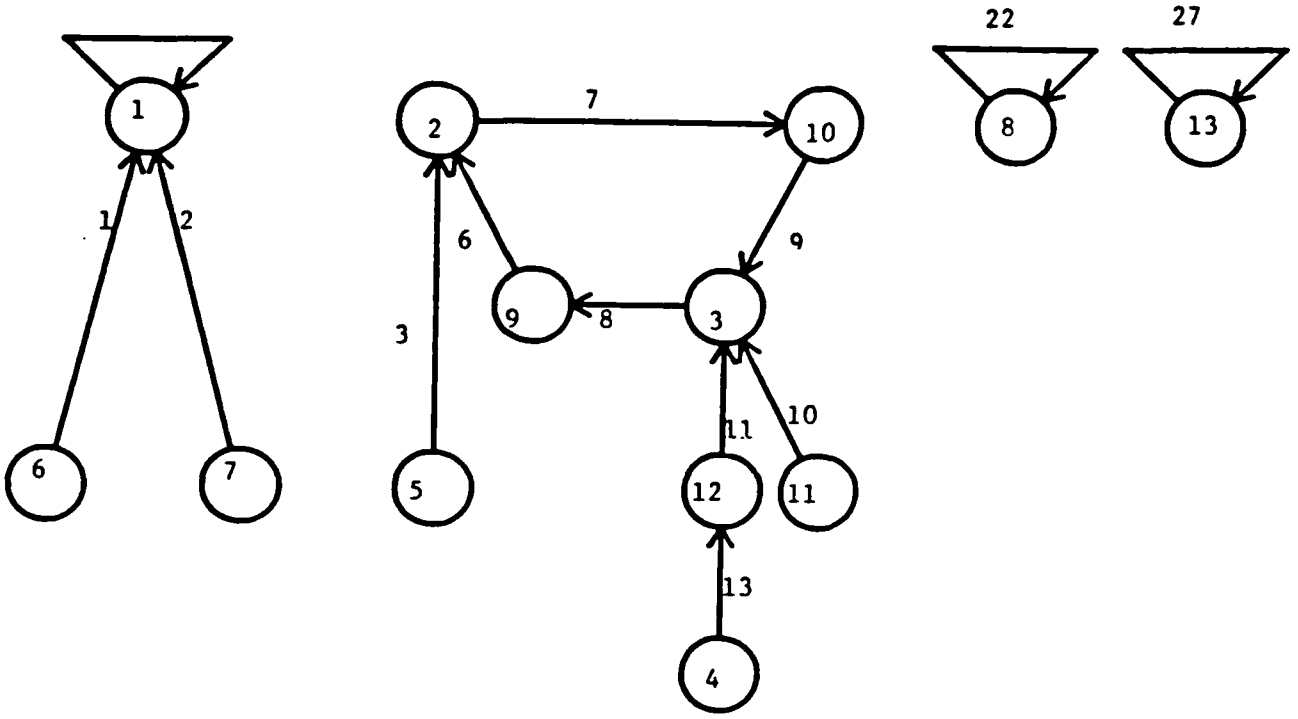
The dual variable (DUAL) for each node is required to determine an arc to enter the basis. Therefore, the dual for each node is retained at all times.

4.2.7 Cycle Multiplier

The cycle multiplier (CMULT) is defined as follows: give an orientation to the cycle; the cycle multiplier is the product of the arc multipliers for arcs in the same direction as the orientation divided by the arc multipliers of those arcs in the reverse direction.

CMULT is defined only for nodes on cycles that are not self-loops. CMULT is the same for all nodes on the same cycle.

The cycle multiplier can be thought of as the amount of flow that



label on arc: arc number
direction on arc: predecessor

Node	Arc
1	15
2	7
3	8
4	13
5	-3
6	-1
7	-2
8	22
9	-6
10	-9
11	-10
12	-11
13	-27

Figure 4-8. Arc Structure

will result if one unit of flow is sent around the cycle in the given orientation. In a valid basis, CMULT cannot be one.

CMULT is needed to determine the arc to exit the basis, and to update the flows. Because of the number of multiplications and divisions required, this is a very time-consuming number to calculate. Fortunately, this number must be calculated just once for each new cycle created. So pivots that do not create a new cycle do not require the calculation of any CMULT values.

5. Handling Arcs

In general there are far more arcs than nodes in a network model. The handling of the arcs is therefore critical to decreasing computation time and storage requirements. Despite the large amount of arc data, very little data is required for any individual pivot. This permits the storage of the arc data outside central computer memory, normally on a high speed mass storage device.

Methods for handling the arc data was reported in detail in [84-09]. The following sections review the conclusions of that report.

5.1 Choosing an Entering Arc

The primal simplex method provides flexibility in the choice of arc to enter the basis. The only property the entering arc must have is that placing a small amount of flow on the arc will decrease the objective function value. It is not necessary to select the arc that will yield the greatest decrease in the objective value the most.

Given the current dual values, it is easy to determine the effect of making a small change in the current flow of a non-basic arc. If the arc currently has no flow on it, increasing the flow by one unit would add the following amount to the objective function:

MULT * DUAL(HEAD) - DUAL(TAIL) - COST

where the arc in question is from node TAIL to node HEAD and has multiplier MULT and cost COST.

If the current flow on the arc is the capacity of the arc, then decreasing the flow by one unit would add the following amount to the

objective:

- (MULT * DUAL(HEAD) - DUAL(TAIL) - COST)

It might not be possible to change the flows by one unit. Some arc in the basis could reach one of its bounds before one unit of flow is placed on the entering arc. Conversely, it might be possible to change the flow by more than one unit. The actual amount of change is determined in Step 3 of the primal simplex algorithm.

The above equations are referred to as the reduced cost for the arc. If the reduced cost is negative, the objective value would be decreased if the non-basic arc were to enter the basis. Non-basic arcs with negative reduced cost are eligible to enter the basis. If there is no arc with a negative reduced cost then the current solution is optimal.

Generally, there are many arcs that are eligible to enter the basis at each iteration. It is necessary to choose from among those possible. For instance, it is possible to calculate all of the reduced costs and choose the arc with the most negative reduced cost to enter the basis. This, generally, will have fewer pivots than other methods but the amount of time required to calculate all of the reduced costs would be prohibitive.

Another alternative is to calculate the reduced costs, one at a time, and choose an arc to enter the basis as soon as one is found with a negative reduced cost. This will minimize the amount of time to calculate reduced costs; but it will cause many pivots which improve, only marginally, the objective value.

In practice, two methods are used to choose an arc to enter the basis. These methods are referred to as the fixed page method and the candidate list method.

5.1.1 Fixed Page Method

In the Fixed Page method, the reduced cost for a fixed number of arcs (a page) is calculated at each pivot. The arc with the most negative reduced cost is then selected to enter the basis. If no arc has a negative reduced cost, then a new page (of arcs) is used. The page size (number of arcs in the page) is important. Too small a page will cause too many pivots; too large a page will cause too much time for the reduced cost calculation.

After each pivot, a decision must be made whether to use the same page of data or to obtain a new page. One method for making this decision is to provide a re-use factor, giving the maximum number of times a page can be used before a new page must be selected.

5.1.2 Candidate List Methods

If arcs from a variety of nodes are examined, then, when one of them enters the basis, generally, it will not effect the reduced cost of many of the other arcs. It is therefore possible to examine only a subset of arcs, called a candidate list. The first step is to create a list of arcs with a negative reduced cost. The arc with the most negative reduced cost is then selected to enter the basis. The reduced costs of the arcs in the list are then recalculated, and the next arc to enter the basis is selected from the list. After a fixed number of pivots, the candidate list is reformed.

Two parameters are required: the candidate list size and the number of iterations before reforming the list.

5.2 Storing Arc Data

In both major methods for choosing an entering arc, only a small amount of arc data is required at any given time. Information on arcs in the basis is always required, but only a small number of non-basic arcs are needed. In the fixed page method, only arcs in the page being examined are needed. In the candidate list method, only arcs in the candidate list are required.

This suggests maintaining arc data on a high speed mass storage device (e.g. hard disk on a microcomputer). Only basic arc information and a small number of search pages is maintained in core. When new arc data is required a page of arcs can be read in, replacing the previous pages. The amount of data remains constant while the actual data is constantly changing. This is called the "in-core/out-of-core" method.

Some method is needed to store information on the non-basic flows. The non-basic flows are either zero or the arc's upper bound. The various possibilities were given in [3]. If there is a large number of arcs then the information on the non-basic flows must also be stored outside central memory. This is slow, but it permits even small computers to solve extremely large problems.

5.3 Specialization for MRMATE

The number of arcs in a MRMATE model can be very large. The largest problems can have more than one-half of a million arcs. Problems of this size require the in-core/out-of-core method.

One advantage of the MRMATE problem is that the structure of the arc costs is known. There will be many arcs with zero cost. These arcs are likely to be in the optimal basis. Therefore, it seems reasonable to enter zero cost arcs as often as possible.

The remaining arcs can be separated into two classes: low cost and high cost arcs. By separating the arcs into three different files (zero, low, and high cost files) arcs with zero cost can be preferentially entered, without calculating reduced costs for low and high cost arcs. To ensure optimality all the arcs must be examined; however, more time can be spent with the zero cost arcs.

Slack and surplus arcs are also very important in the solution process. These arcs should be examined more often than other arcs. If the "wrong" slack and surplus arcs are in the basis many pivots might be performed unnecessarily. These arcs should not be kept out of core. Information necessary to generate these arcs should be available in core and their reduced costs should be recalculated frequently.

6. Finding the Exiting Arc

In the simplex method specialized for generalized networks, only a limited number of arcs are candidates to exit the basis during any pivot. The rapid identification of these arcs is a reason the specialized method is more efficient than the simplex method for general linear programming.

For any node, NODE, define the "backpath" of NODE to be those arcs between NODE and the cycle for the component that contains NODE, as well as those arcs on the cycle. In other words, the backpath for NODE contains those basic arcs whose corresponding node can be reached from NODE by use of the PRED structure only. For the entering arc number ARC, the only arcs to change flow are those on the backpaths of TAIL(ARC) and HEAD(ARC). Figure 6-1 shows the arcs that will change flow for various combinations of HEAD(ARC) and TAIL(ARC), referred to as HEAD and TAIL respectively.

If a single unit of flow were placed on the arc entering the basis, the flows on the arcs in the backpaths are the only ones which must be updated so as to keep the net flow at each node the same. For instance, the arc between TAIL and PRED(TAIL) must provide one unit of flow at node TAIL. If that arc is oriented from PRED(TAIL) to TAIL and has multiplier of MULT, then the arc between PRED(PRED(TAIL)) and PRED(TAIL) must provide 1/MULT units of flow at PRED(TAIL), and so on (see Figure 6-1).

This calculation is equivalent to determining the updated column in linear programming. With this updated column, it is possible to determine the amount of flow by which the entering arc flow can

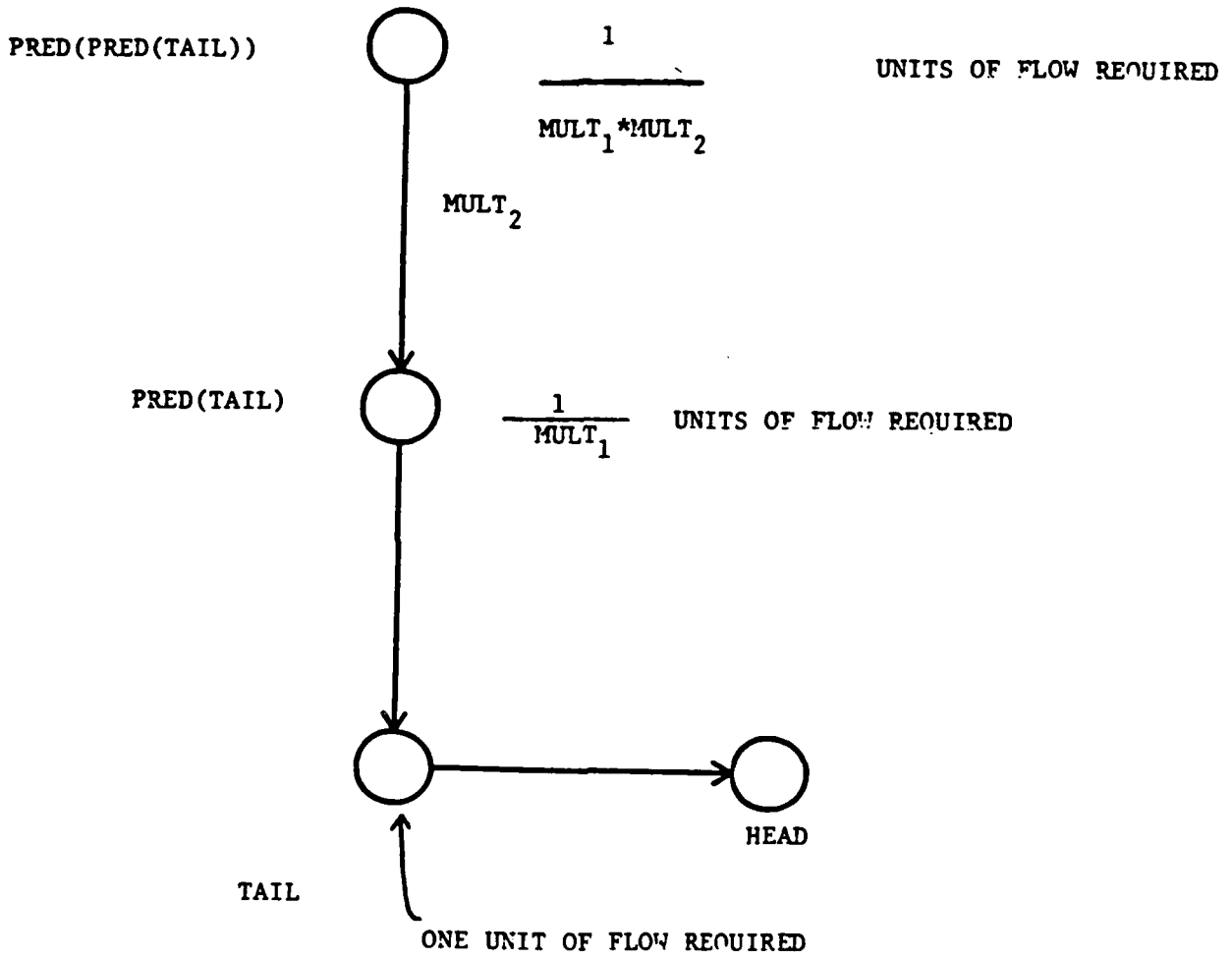


Figure 6-1. Flow Required at Nodes

change before some basic arc reaches one of its bounds. The first arc to reach one of its bounds is the exiting arc from the basis.

6.1 Calculating the Exiting Arc

The determination of the exiting arc involves two steps: the development of the updated column and the determination of the maximum flow change on the entering arc, ARC (before a basic arc reaches one of its bounds). These two steps can be performed simultaneously for each node. For simplification, the presentation here will separate the process. There are several cases to consider.

The first case assumes that the backpaths from TAIL and HEAD are distinct, implying that TAIL and HEAD are in different components. If the multiplier on ARC is MULT then placing one unit of flow on ARC will require one unit of flow at TAIL and will provide MULT units of flow at HEAD. The sign convention of a negative number for a demand and a positive number for a supply will be adopted. The value of -1 for TAIL and MULT for HEAD is called the requirement.

Given the requirement at a node NODE, two pieces of information are required: the update column entry for the arc between NODE and PRED(NODE) and the requirement for PRED(NODE).

These values depend on the orientation of ARC(NODE), the basic arc between NODE and PRED(NODE). The following algorithm calculates the entry in the updated column (UP_COL) and updates the requirement for NODE (REQUIRE) to be the requirement for PRED(NODE).

```
if ARC(NODE) < 0 then (*arc oriented from PRED(NODE) to NODE *)
     REQUIRE := REQUIRE/MULT(NODE);
     UP_COL(NODE) := REQUIRE;
else
     UP_COL(NODE) := -REQUIRE;
```

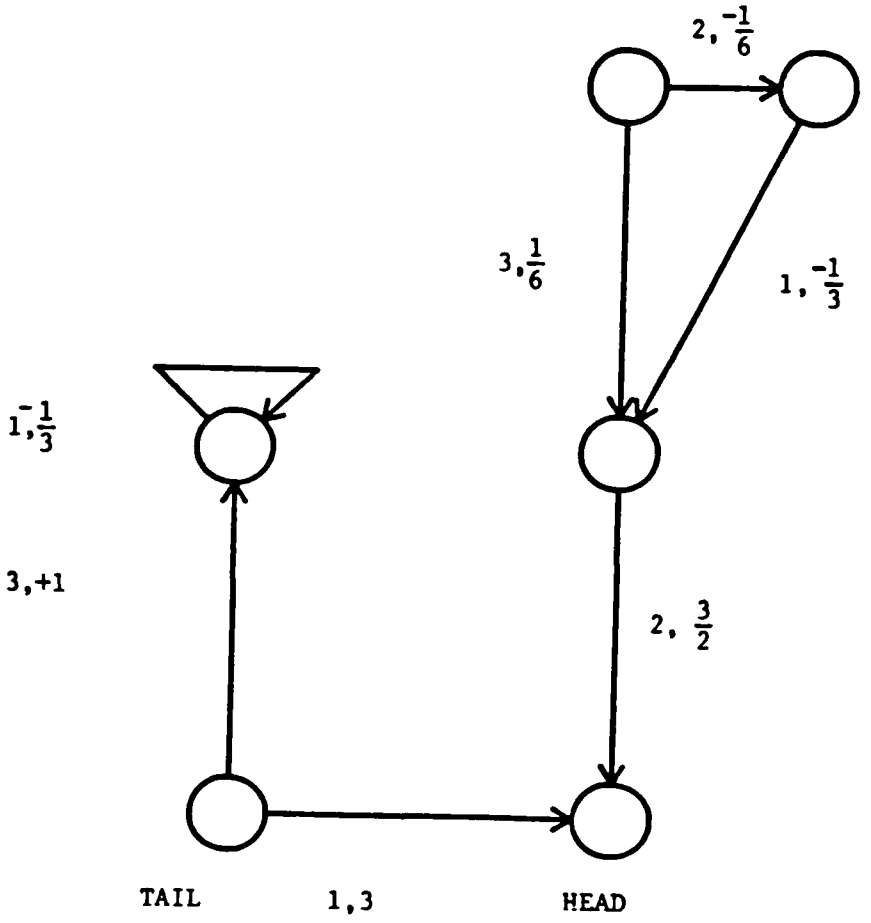
```
        REQUIRE := REQUIRE * MULT(NODE);
    endif
```

The process is slightly more complicated when a cycle, not a self-loop, is reached. The effect of the cycle multiplier must be taken into effect. Briefly, the cycle is used to create or destroy flow as needed. The CMULT value determines the rate at which flow can be created or destroyed. The effect of the cycle multiplier is that when the cycle is reached, REQUIRE is replaced by REQUIRE / (1-CMULT(CYCLE)). The equations above can then be used for the arcs on the cycle. These calculations can be carried out independently for the backpaths of HEAD and TAIL when the nodes are in different components.

For the second case, when the two nodes are in the same component, the backpaths will coincide at some point. The updated column is the sum of the updated columns calculated using the above equations. If the backpaths coincide before the root cycle it is possible to simply add together the REQUIRE values of the two backpaths and continue as though only one backpath existed. If the value of REQUIRE is zero (as it will be for pure networks) no further calculations need be done; the rest of the arcs will not change in flow. If the backpaths coincide only on the cycle, it is easiest to proceed around the cycle twice making the necessary calculations and add together the resulting updated columns.

An example of these calculations is given in Figure 6-2.

Given UP_COL(NODE) it is possible to determine the amount of change permitted on the entering arc before the arc associated with NODE reaches a bound. Let INCREASE := 1 if the entering arc is



label: MULT, UP_COL

Figure 6-2. Updated Column

currently at zero flow and -1 if the entering arc has flow equal to its capacity. The calculation of the maximum change is as follows:

```
if (INCREASE * UP_COL(NODE) > 0) then
     (* flow will decrease on ARC(NODE) *)
     MAX := FLOW(NODE) / UP_COL(NODE);
else
     (* flow will increase on ARC(NODE) *)
     MAX := (CAPACITY(NODE) - FLOW(NODE)) / UP_COL(NODE);
endif
```

By taking the minimum value for MAX over all nodes with a changing flow, the flow change on the entering arc is determined and the exiting arc is identified. If this value is more than the CAPACITY of the entering arc then the change in flow is the CAPACITY of the entering arc and the exiting arc is the entering arc.

7. Updating the Basis

Given the arc to enter and and the arc to leave the basis, the final step is to update the basis structures. This update requires updating the DUALs, the FLOWs and the rest of the basis structures.

7.1 Updating the Basis Structures

7.1.1 Pivot Types

The linked rooted tree method has six pivot types, depending on the relationship between the entering and exiting arc.

Pivot type 1 occurs when the entering arc and exiting arc are the same. This occurs when flow on the entering arc reaches its upper bound or zero before any other flows reach their limits. In this case, the basis remains the same, so only FLOWs must be changed.

When TAIL and HEAD of the entering arc are in the same tree, the pivot type is defined to be either 2, 3 or 4. Consider paths from the entering tail node to the cycle and from the entering head node to the cycle. The first node that occurs on both paths is called the meeting node (MEET). The (common) cycle node is called CYC. The exiting arc can occur in three places: before MEET, between MEET and CYC, and after CYC. These three places correspond to pivot types 2, 3 and 4 respectively.

When TAIL and HEAD of the entering arc are in the same component, the pivot type is 2 or 5. If the exiting arc is on the cycle, the pivot type is 5. Otherwise it is type 2.

If TAIL and HEAD of the entering arc are in different components, the pivot type is 6 if the exiting arc is on a cycle, and is 2 otherwise.

Figure 7-1 gives examples of all of these pivot types.

7.1.2 Common Routines

The main advantage of the linked rooted tree method of storing the generalized network basis is that the basis update routines involve a limited number of tree manipulation routines. Each pivot type uses these routines in a different way to create the new basis.

There are five tree manipulation routines required:

1) Hang tree (HANG) : Takes two trees and a node within the first tree and attaches the second tree to the first below the node.

2) Isolate subtree (ISOLATE): Takes a tree and a node within the tree and isolates the subtree below the node by creating a new tree.

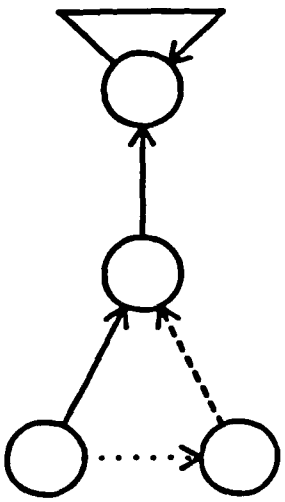
3) Reroot tree (REROOT) : Takes a tree and a node within the tree and makes that node the root of the tree.

4) Reverse cycle reroot (REV_CYC_REROOT): Takes a series of trees connected by PRED values and creates a new tree consisting of all of them in reverse order.

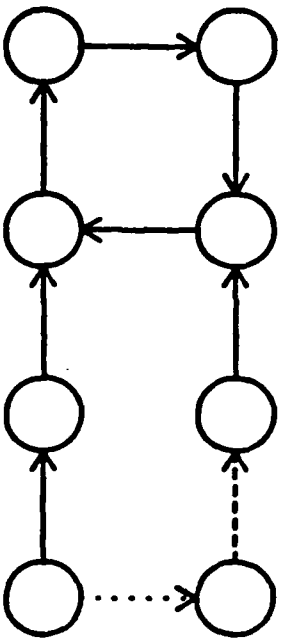
5) Cycle reroot (CYC_REROOT): Takes a series of trees connected by PRED values and creates a new tree consisting of all of them.

7.1.2.1 HANG Routine

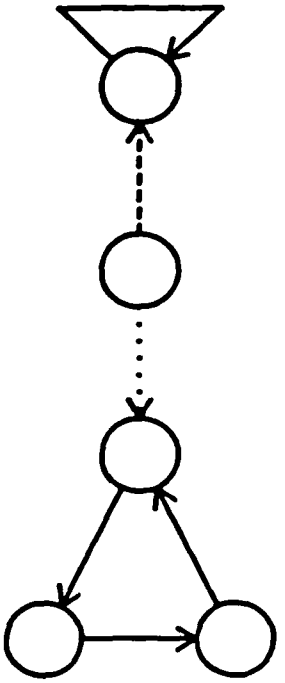
INPUT:



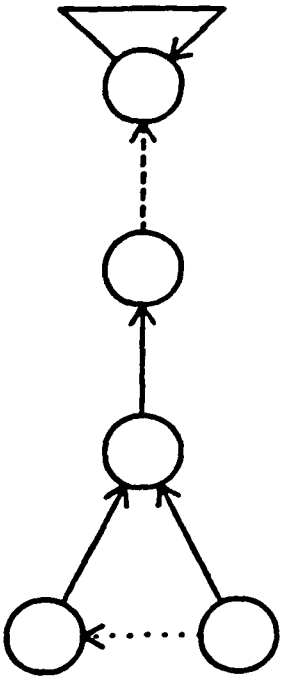
(a) pivot type 2



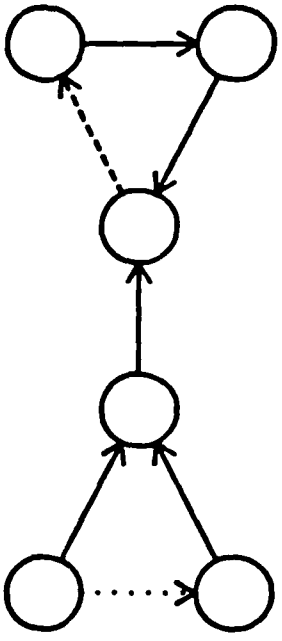
(b) pivot type 2



(c) pivot type 2



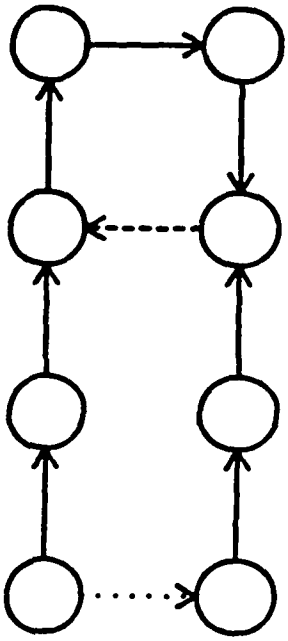
(d) pivot type 3



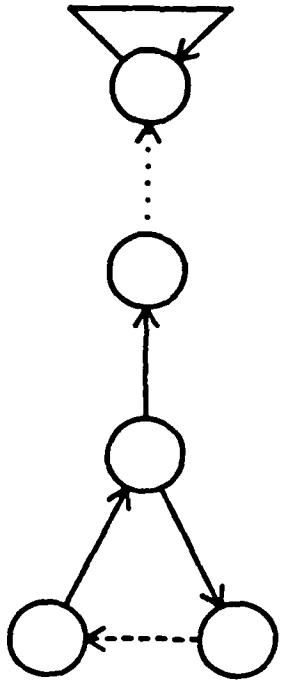
(e) pivot type 4

Key: ... entering arc
--- exiting arc
___ other basic arcs

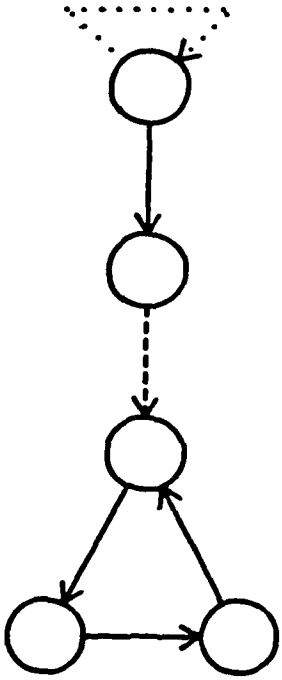
Figure 7-1. Pivot Types



(f) pivot type 5



(g) pivot type 6



(h) pivot type 7

Figure 7 -1 Pivot Types (cont.)

ROOT: a node, not necessarily the root of one tree;

TREE: the root node of another tree.

OUTPUT:

Updated PRED, THREAD, LEVEL and RTHREAD for a tree with the nodes of TREE below ROOT. (See Figure 7-2).

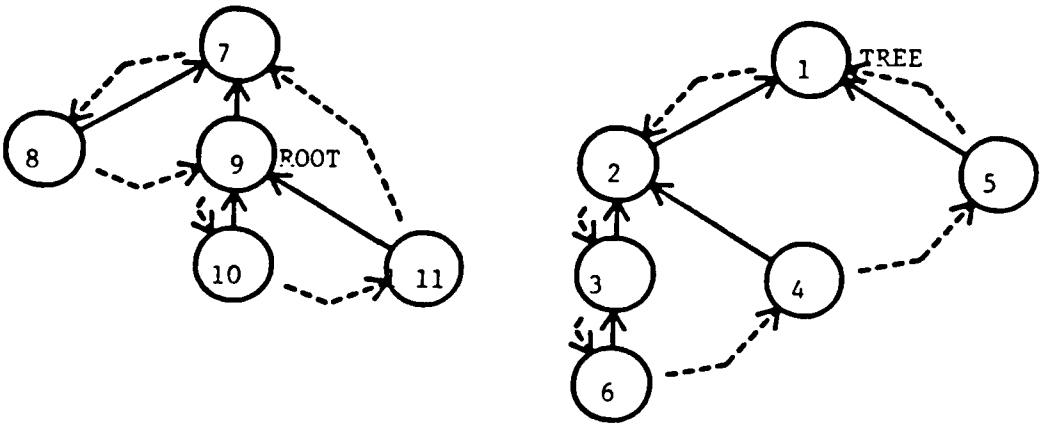
METHOD:

```
(A)  Find the final node in preorder traversal (LAST) in the
     subtree below ROOT by following the THREAD values;

(B)  LEVEL(TREE) := LEVEL(ROOT) + 1;
     NODE := THREAD(TREE);
     while NODE <> TREE do
          LEVEL(NODE) := LEVEL(PRED(NODE)) + 1;
          NODE := THREAD(NODE);
     endwhile;

(C) TEMP := THREAD(LAST);
    THREAD(LAST) := TREE;
    THREAD(RTHREAD(TREE)) := TEMP;
    RTHREAD(TEMP) := RTHREAD(TREE);
    RTHREAD(TREE) := LAST;
    PRED(TREE) := ROOT;
```

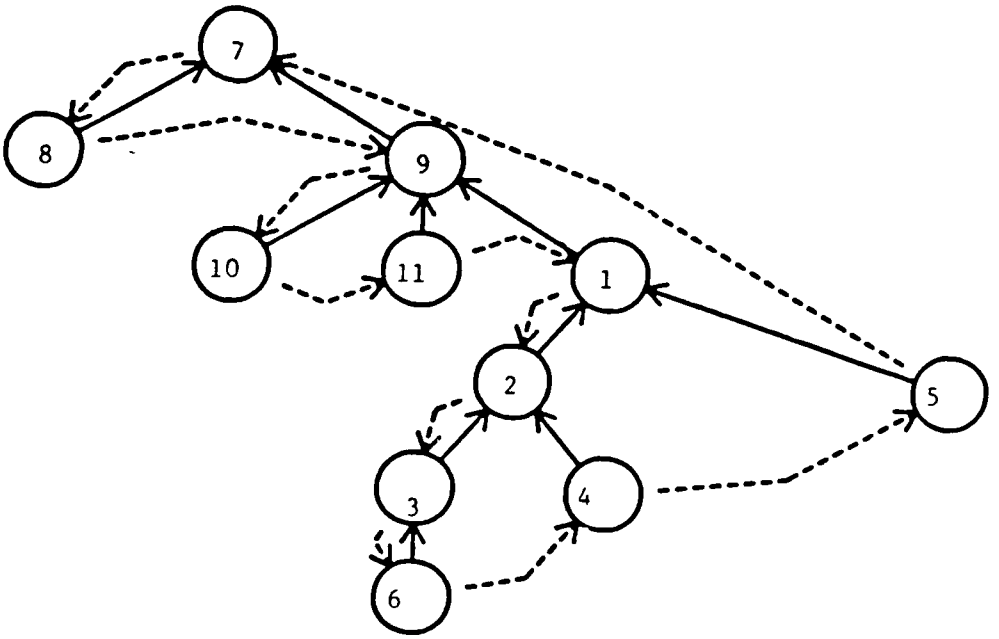
DISCUSSION:

Two common programming structures are exhibited in this routine. In section B, the LEVEL structure is updated for TREE. Because the THREAD is the preorder traversal, the level of PRED(NODE) is always calculated before the level of NODE. Since LEVEL(NODE) is always equal to LEVEL(PRED(NODE)) + 1 within trees, the level calculation is simplified.

The second structure is in section C. For roots of trees, the reverse thread of the root is always the last node in the preorder traversal. This means that finding the last node in TREE is easy, as opposed to the last node below ROOT. Section A is needed to find the



BEFORE HANG



AFTER HANG (9,1)

Figure 7-2. HANG Routine

last node below ROOT.

7.1.2.2 ISOLATE Routine

INPUT:

NEW_ROOT: a node in a tree

OUTPUT:

Updated THREAD and RTHREAD so that nodes below NEW_ROOT form a tree rooted at NEW_ROOT. (See Figure 7-3).

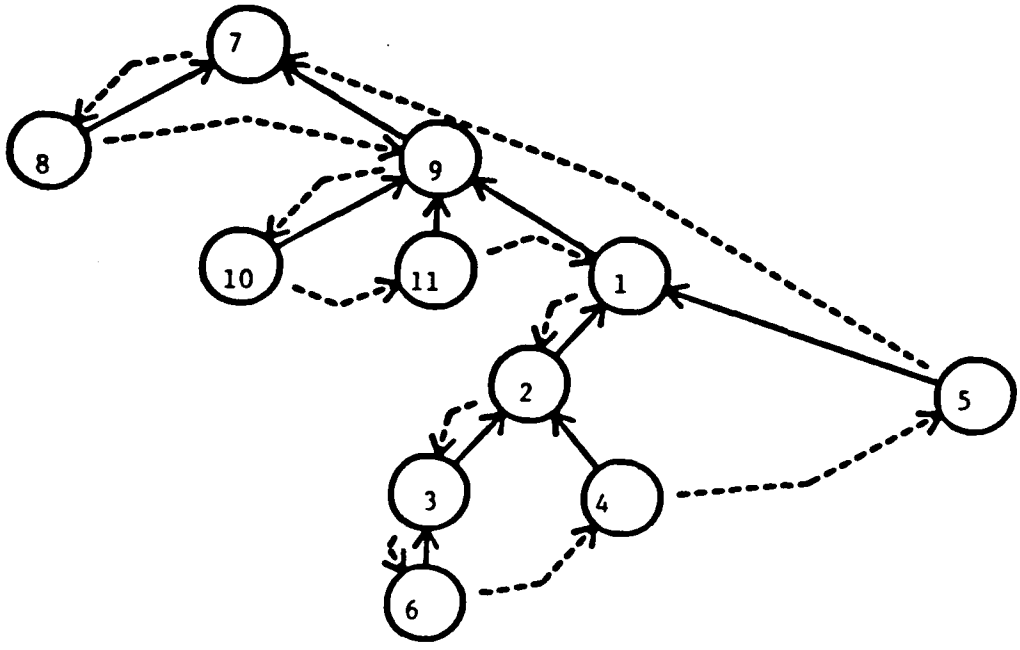
METHOD:

(A) Find the last node (LAST) in the subtree rooted at NEW_ROOT by following the THREAD values;

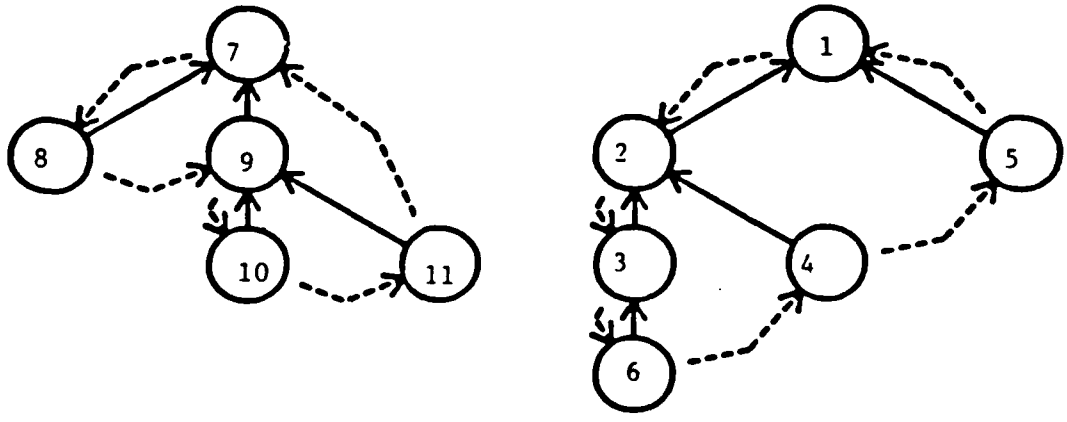
(B)
```
THREAD(RTHREAD(NEW_ROOT)) := THREAD(LAST);
RTHREAD(LAST) := RTHREAD(NEW_ROOT);
THREAD(LAST) := NEW_ROOT;
RTHREAD(NEW_ROOT) := LAST;
```

DISCUSSION:

Once the last node in the subtree below NEW_ROOT has been located, rethreading involves only nodes LAST and NEW_ROOT.



BEFORE ISOLATE



AFTER ISOLATE (1)

Figure 7-3. Isolate Routine

7.1.2.3 REROOT Routine

INPUT:

TREE: the root node of the subtree to be rerooted;

NEW_ROOT: a node in the tree rooted at TREE.

OUTPUT:

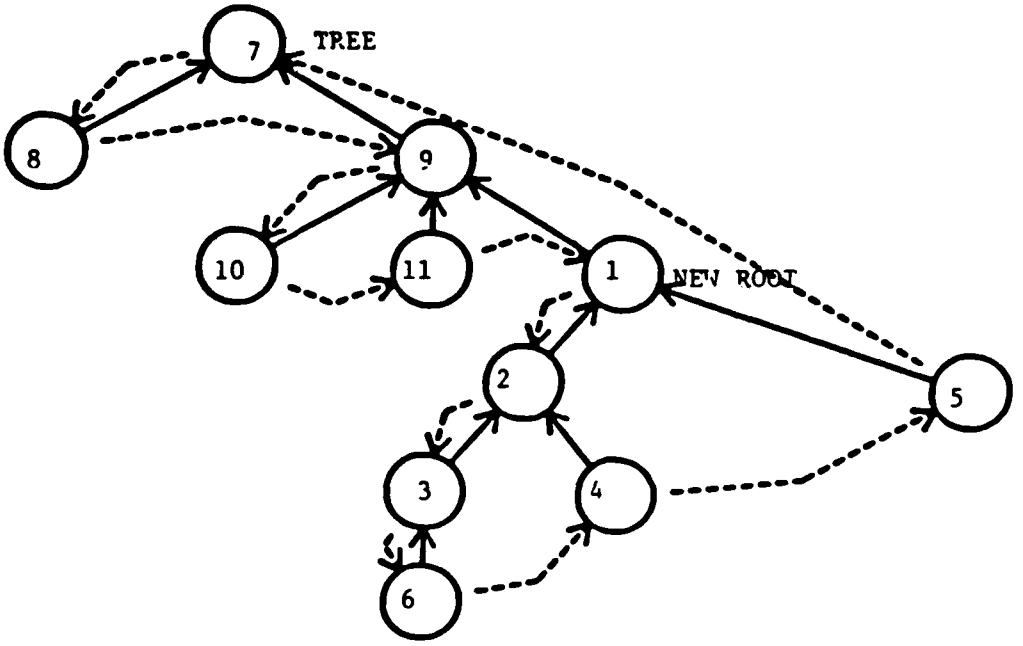
Updated PRED, THREAD, LEVEL and RTHREAD for a tree consisting of the nodes below TREE, with root, NEW-ROOT. (See Figure 7-4).

METHOD:

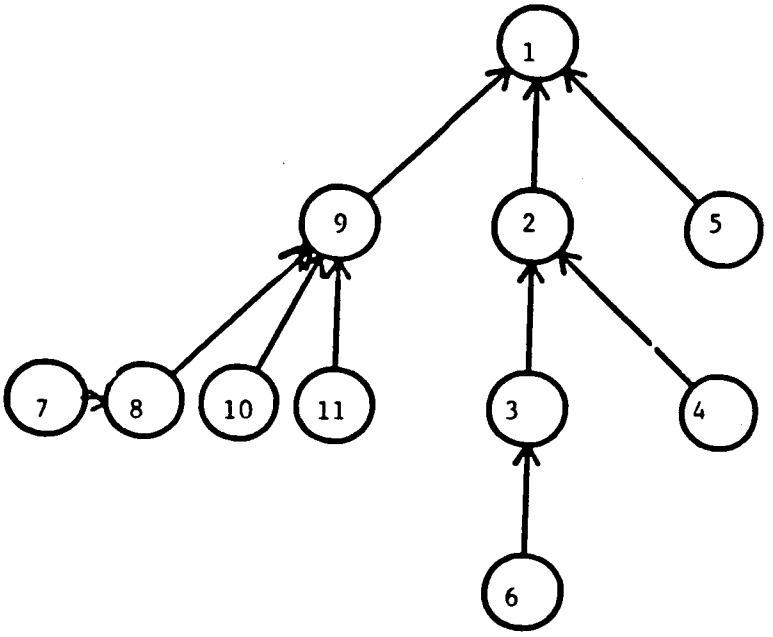
```
LEVEL(NEW_ROOT) := 0;
NODE := NEW_ROOT;
PREV := NEW_ROOT;
while PREV <> TREE do
     ISOLATE(NODE);
     TEMP := PRED(NODE);
     if NODE <> NEW_ROOT then
          HANG(PREV,NODE);
     endif;
     PREV := NODE;
     NODE := TEMP;
endwhile;
```

DISCUSSION:

Using HANG and ISOLATE, rerooting a tree at a new node is a simple task. Each node on the path from NEW_ROOT to TREE is isolated and then hung from the previous node. The temporary variable TEMP is required because HANG changes the value of PRED(NODE).



BEFORE



After Reroot (7,1)

Figure 7-4. Reroot Routine

7.1.2.4 REV_CYC_REROOT Routine

INPUTS:

FIRST_TREE: First tree of the sequence to put together;

LAST_TREE: Last tree of the sequence;

OUTPUTS:

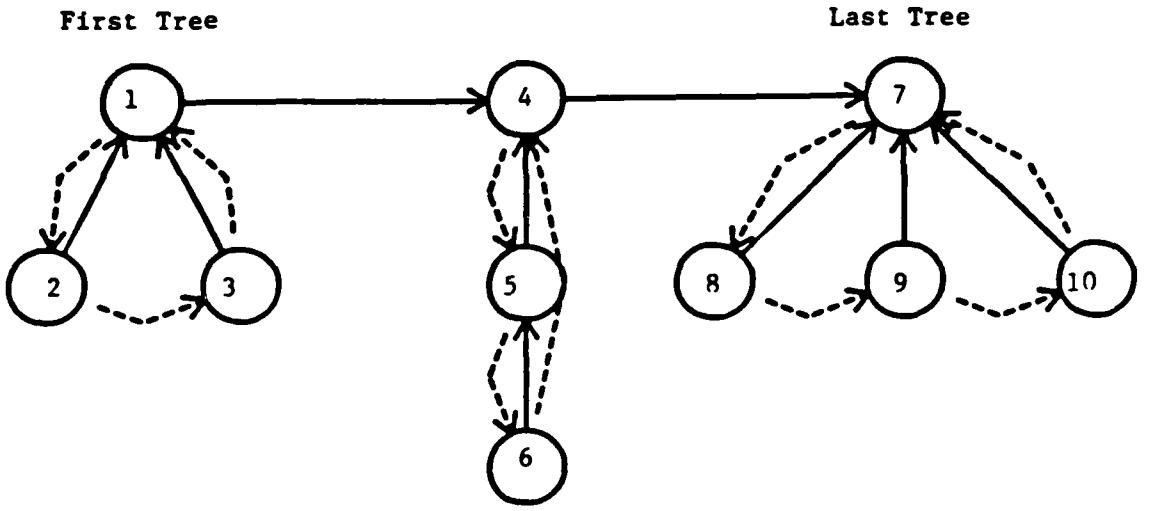
Updated PRED, THREAD, RTHREAD and LEVEL values for a single tree, rooted at FIRST_TREE, consisting of trees rooted at predecessor values from FIRST_TREE to LAST_TREE. Note that this definition requires that the predecessors along the path from FIRST_TREE to LAST_TREE be reversed. (See Figure 7-5).

METHOD:

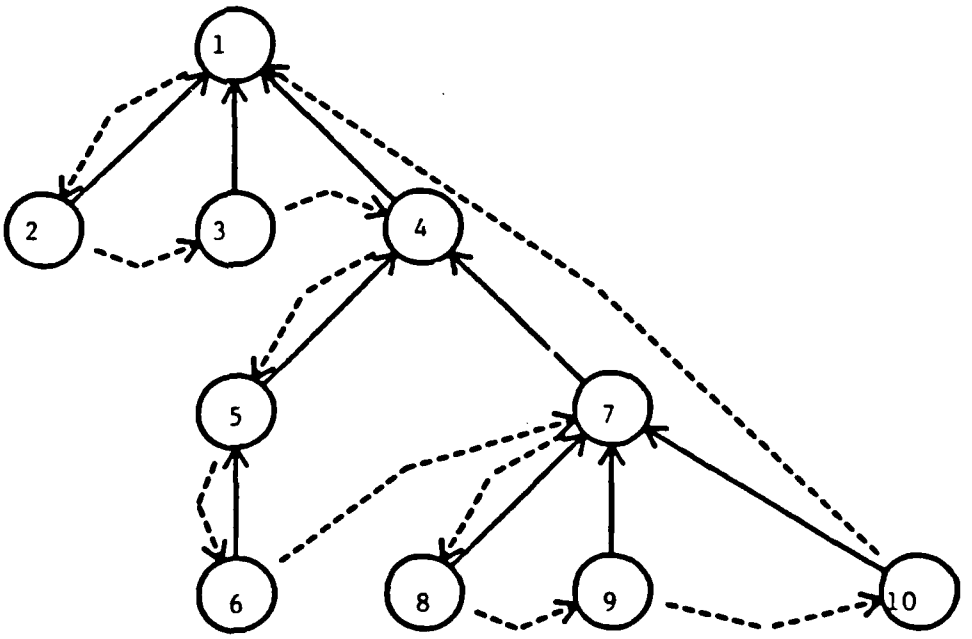
```
NODE := FIRST_TREE;
PREV := FIRST_TREE;
while PREV <> LAST_TREE do
     TEMP := PRED(NODE);
     if NODE <> FIRST_TREE do
          HANG(PREV,NODE);
     endif
     PREV := NODE;
     NODE := TEMP;
endwhile;
```

DISCUSSION:

This routine is essentially the same as REROOT, except that ISOLATE has already been accomplished.



Before



After Rev-Cyc-Reroot (1,7)

Figure 7-5. Rev-Cyc-Reroot Routine

7.1.2.5 CYC_REROOT

INPUTS:

FIRST_TREE: First tree of the sequence to put together;

LAST_TREE: Last tree of the sequence;

OUTPUTS:

Updated PRED, THREAD, RTHREAD and LEVEL values for a single tree, rooted at LAST_TREE, consisting of trees rooted at predecessor values from FIRST_TREE to LAST_TREE. This differs from REV_CYC_REROOT only in that LAST_TREE is the root instead of FIRST_TREE. (See Figure 7-6).

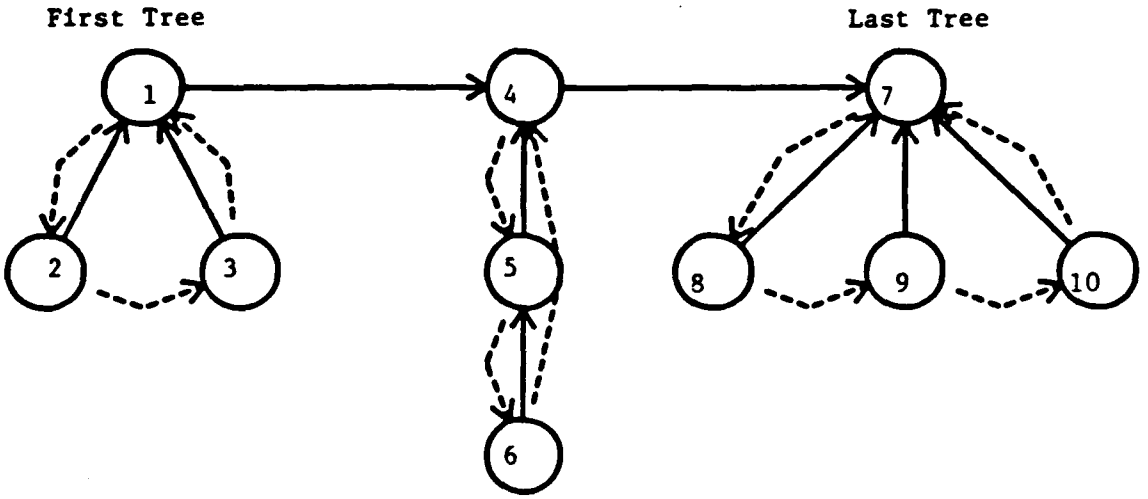
METHOD:

Reverse the PRED values from FIRST_TREE to LAST_TREE;

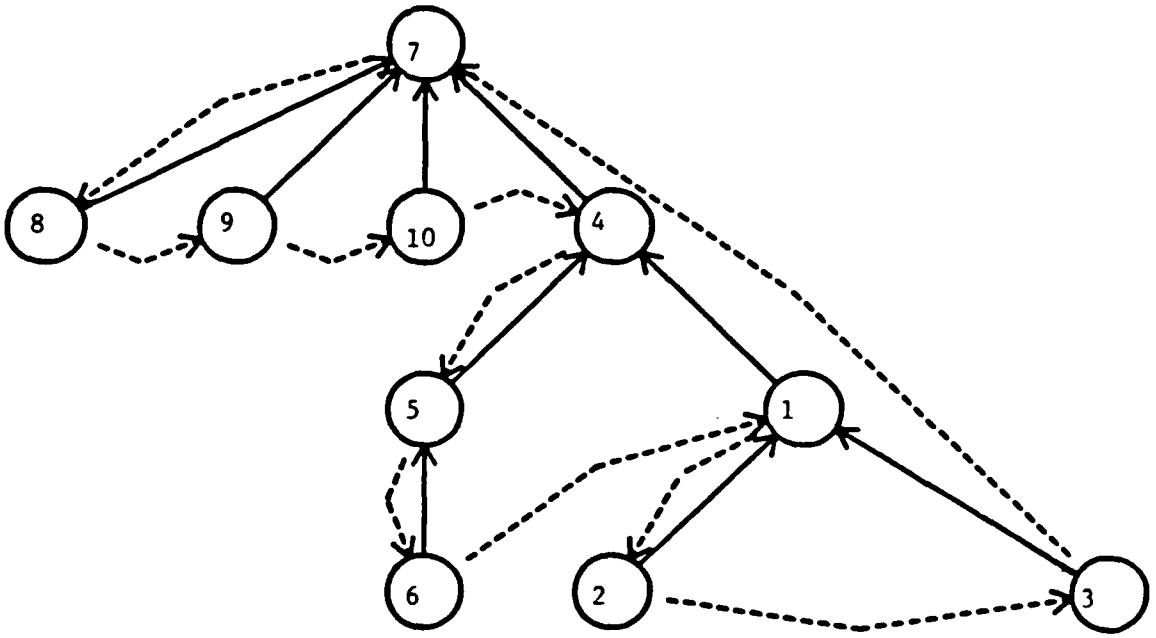
CYC_REROOT(LAST_TREE,FIRST_TREE);

DISCUSSION:

Since CYC_REROOT reverses the PRED values, then by reversing the PRED values before calling CYC_REROOT, the PRED values remain unaffected.



Before



After Cyc-Reroot (1,7)

Figure 7-6. Cyc-Reroot Routine

7.1.3 The Pivot Routines

7.1.3.1 Pivot Type 1

Since the exiting and entering arc are the same, no basis structures other than FLOW must be updated.

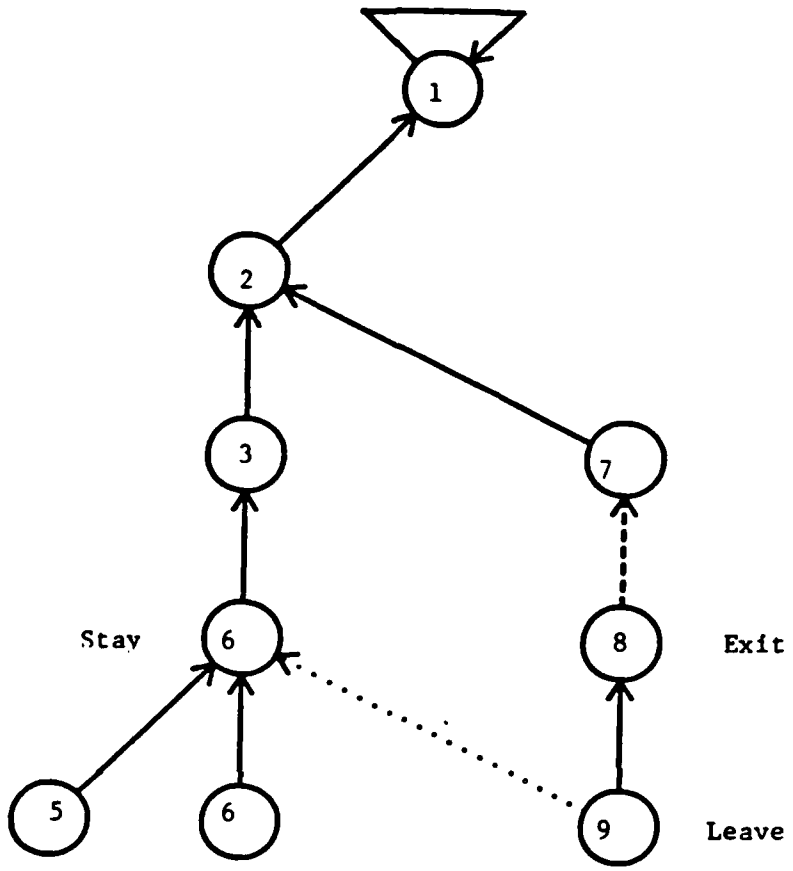
7.1.3.2 Pivot Type 2

Pivot type 2 can occur in one of three cases: entering head and tail are in the same tree, they are in different trees but the same component, or they are in different components. In all cases, the exiting arc is on either the path from the entering tail node to the cycle or the path from the entering head node to the cycle, but not both. The LEAVE node is defined to be the node associated with the path containing the exiting arc. The other node is called the STAY node (see Figure 7-7). EXIT refers to the node that is associated with the exiting arc.

The routine to execute a type 2 pivot is:

```
REROOT(EXIT,LEAVE);
HANG(STAY,LEAVE);
```

See Figure 7-8.



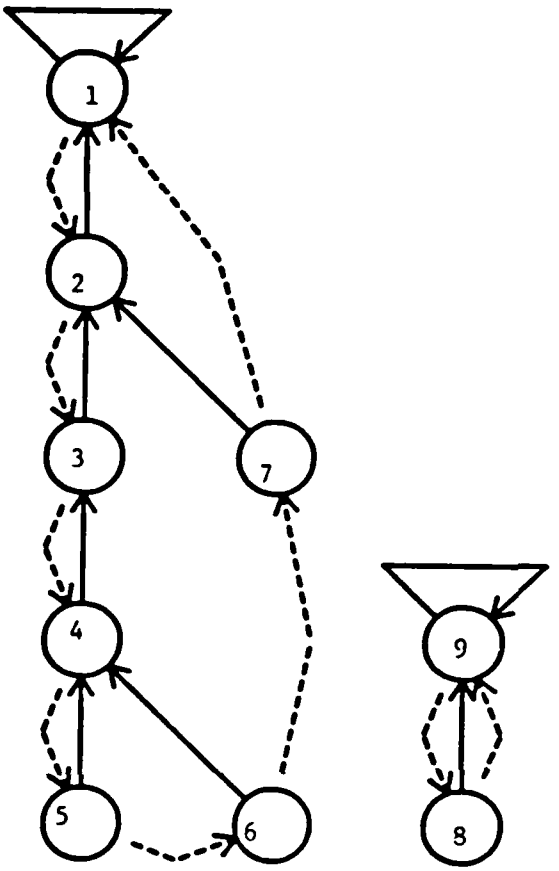
KEY

- - - exiting arc

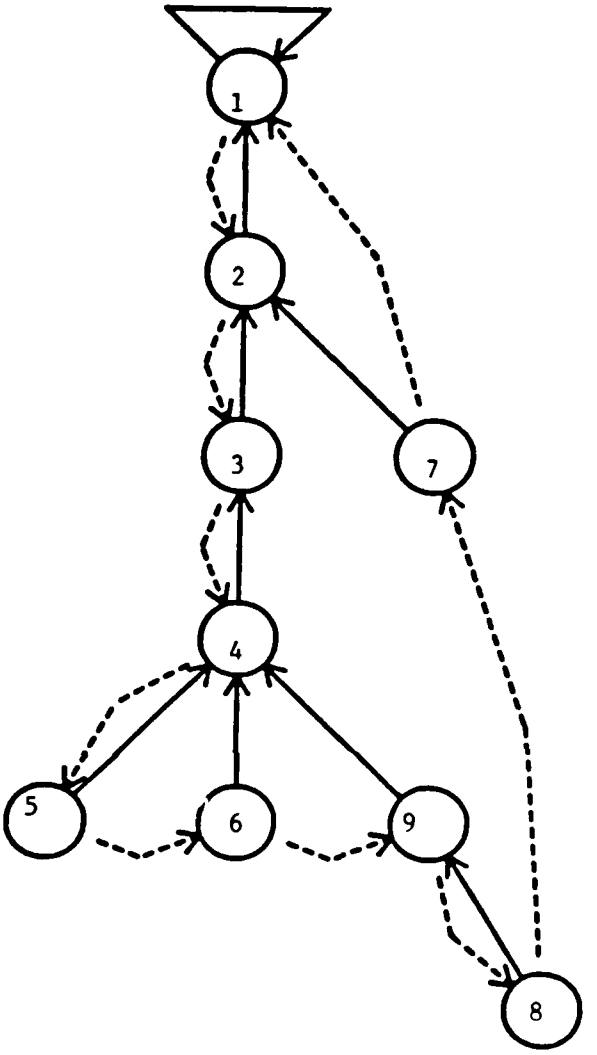
. . . entering arc

_____ predecessor

Figure 7-7. Pivot Type 2 - Initial Position



After Reroot (8,9)



After Hang (4,9)

Key

Predecessor

Thread

Figure 7-8. Pivot Type 2

7.1.3.3 Pivot Type 3

Pivot type 3 occurs when the entering tail and head are in the same tree and the exiting arc is between MEET and CYC (Figure 7-9). Denote the entering tail and head nodes as ETAIL and EHEAD.

The algorithm for pivot type 3 is:

```
NODE := ETAIL;
PREV := EHEAD;
while NODE <> MEET do
     ISOLATE(NODE);
     TEMP := PRED(NODE);
     PRED(NODE) := PREV;
     PREV := NODE;
     NODE := TEMP;
endwhile;

NODE := EHEAD;
PREV := ETAIL;
while NODE <> MEET do
     ISOLATE(NODE);
     NODE := PRED(NODE);
endwhile;
PRED(MEET) := PREV;

REROOT(EXIT,MEET)
```

See Figure 7-10.

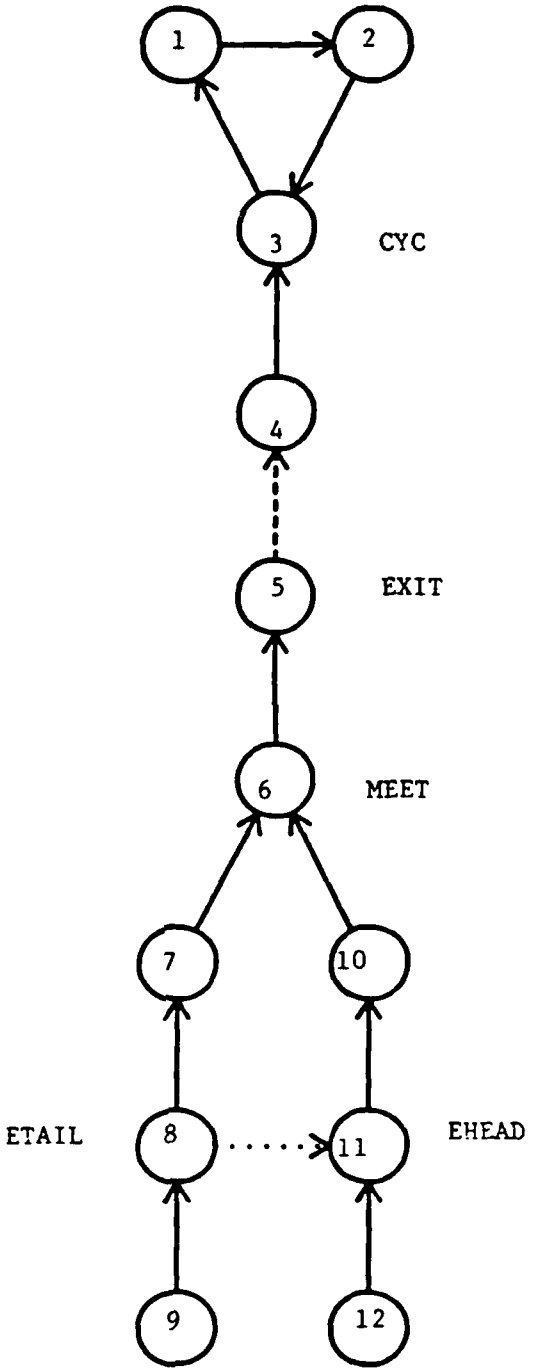
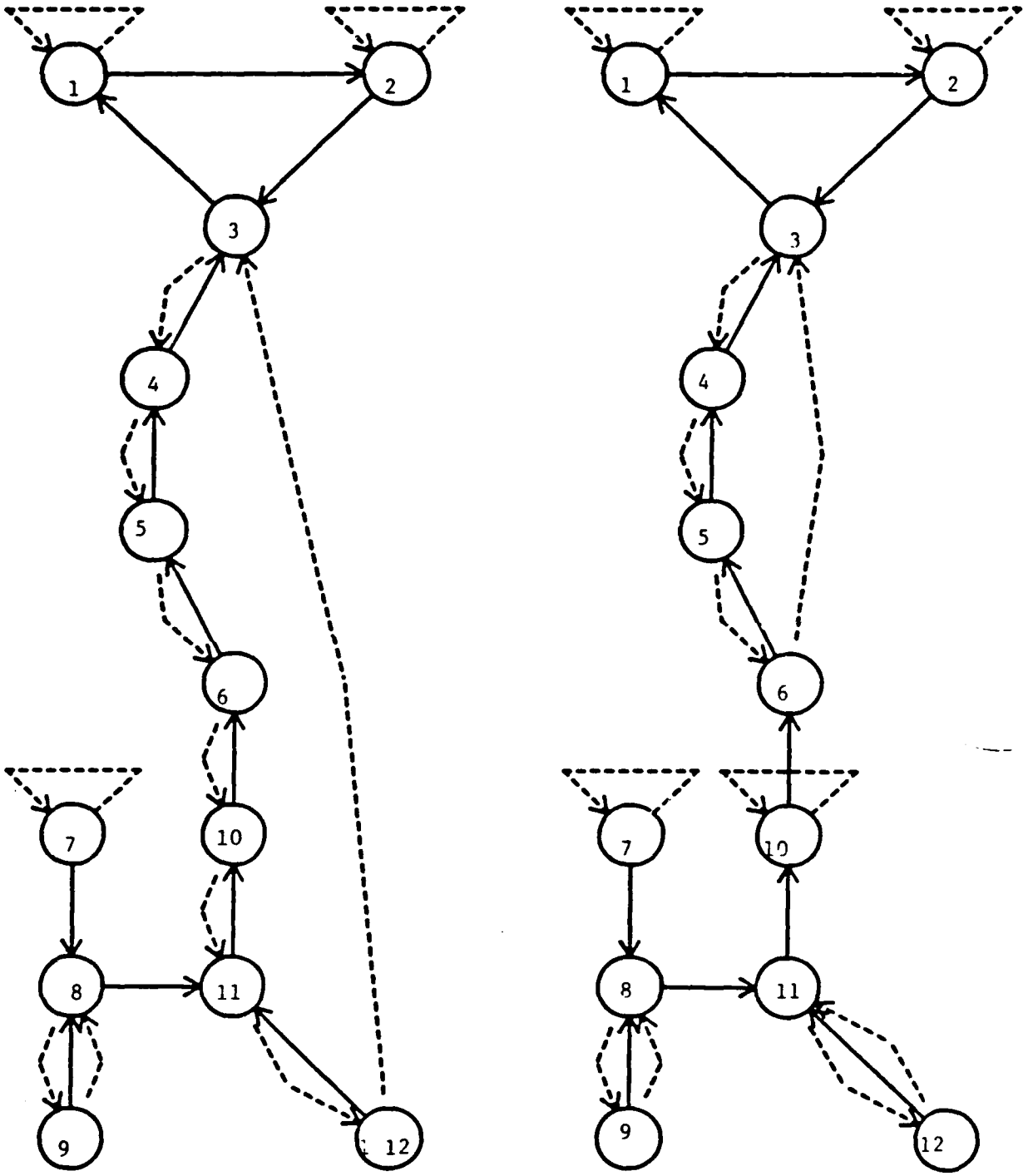


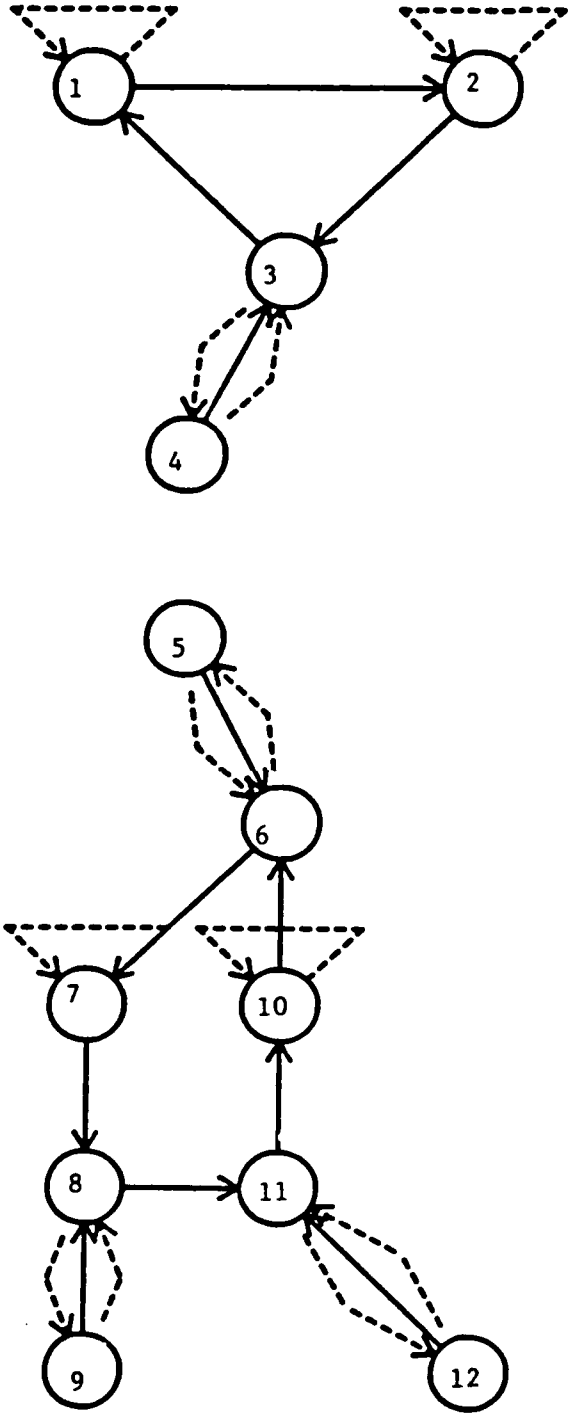
Figure 7-9 Pivot Type 3 - Initial Position



After Section A

After Section B

Figure 7-10. Pivot Type 3 During pivot



After Reroot (5,6)

Figure 7-10. Pivot Type 3 During Pivot (Cont.)

7.1.3.4 Pivot Type 4

Pivot type 4 occurs when the entering head and tail nodes occur in the same tree and the exiting arc is on the cycle (Figure 7-11).

The algorithm for pivot type 4 is:

```
NODE := ETAIL;
PREV := EHEAD;
while NODE <> MEET do
      ISOLATE(NODE);
      TEMP := PRED(NODE);
      PRED(NODE) := PREV;
      PREV := NODE;
      NODE := TEMP;
endwhile;

NODE := EHEAD;
PREV := ETAIL;
while NODE <> MEET do
      ISOLATE(NODE);
      PREV := NODE;
      NODE := PRED(NODE);
endwhile;

PRED_EXIT := PRED(EXIT);
REROOT(CYC,MEET);
REV_CYC_REROOT(CYC,EXIT);
CYC_REROOT(PRED_EXIT,CYC);
PRED(MEET) := PREV;
```

See Figure 7-12.

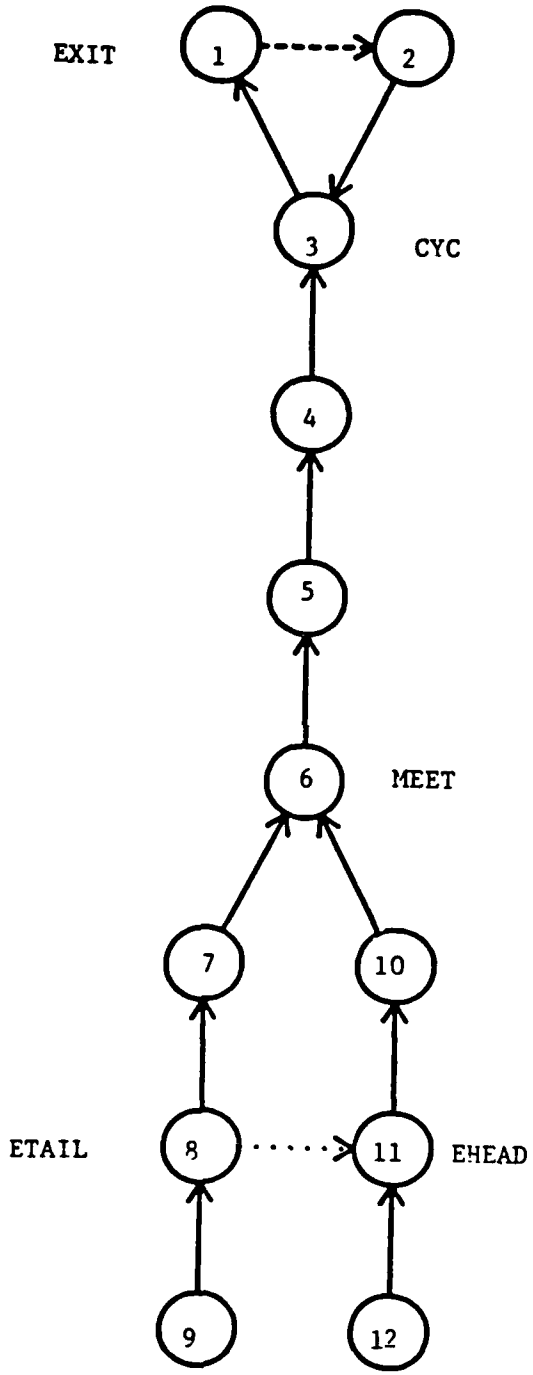


Figure 7-11. Pivot Type 4 - Initial Position

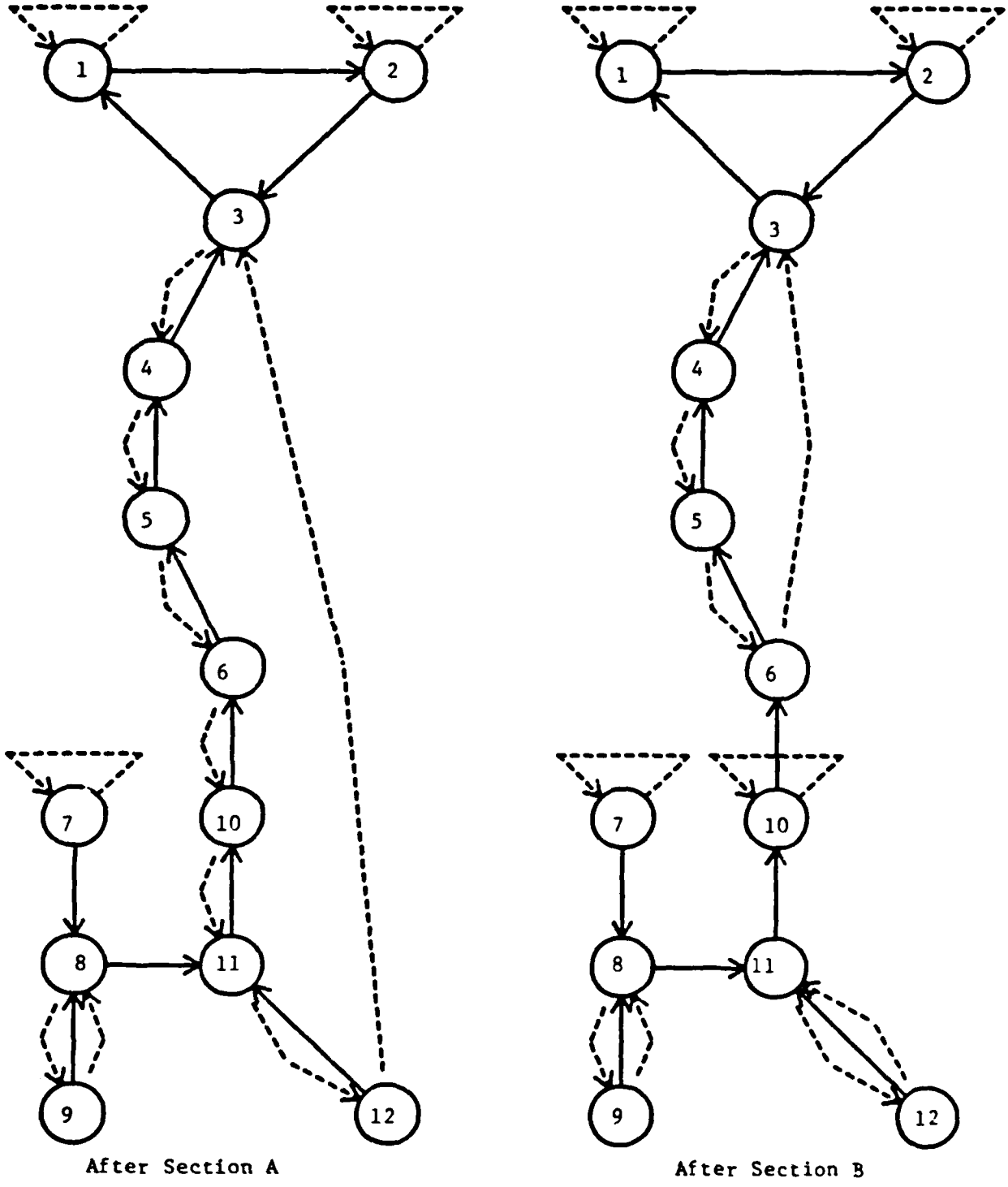


Figure 7-12. Pivot Type 4 - During Pivot

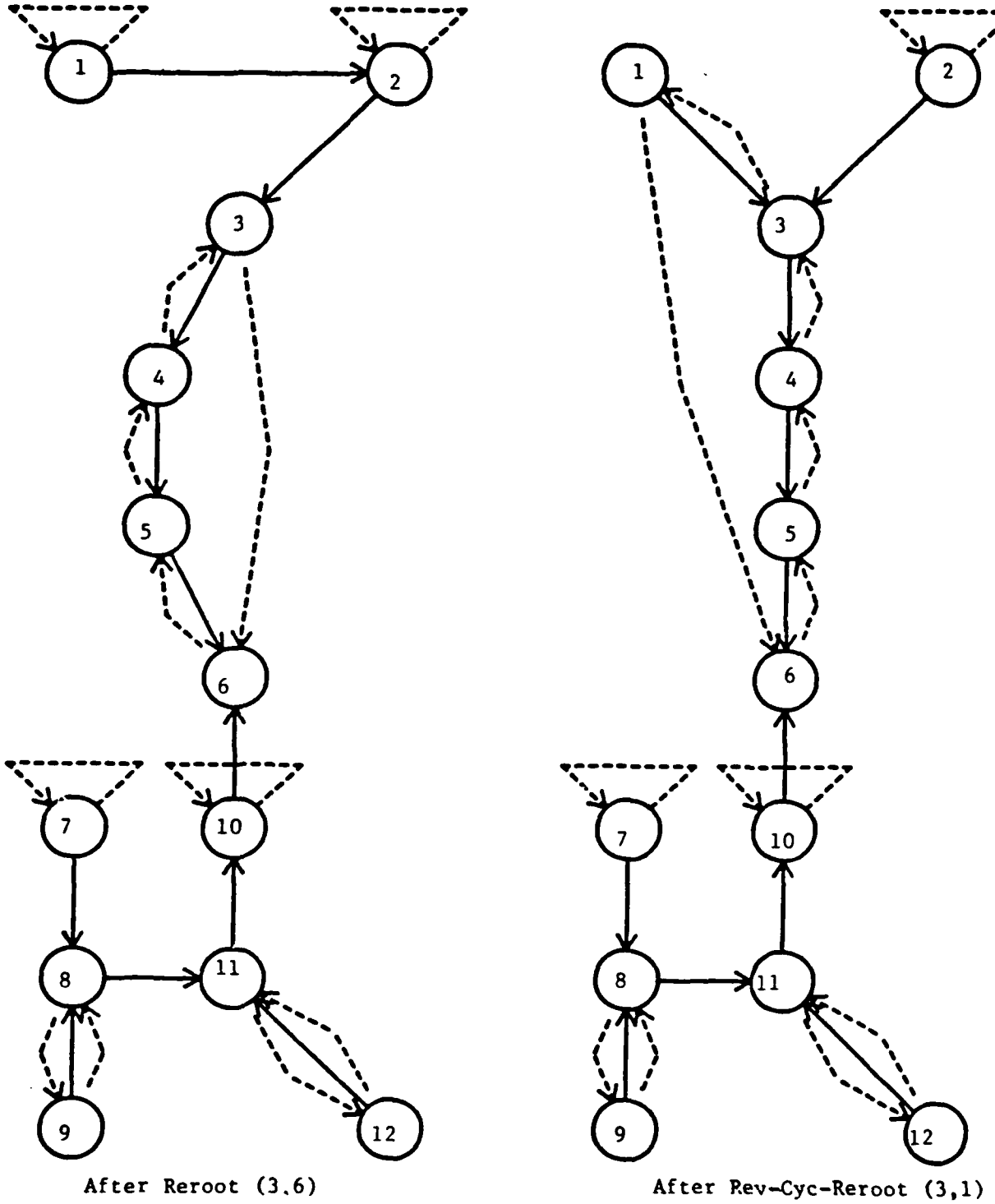
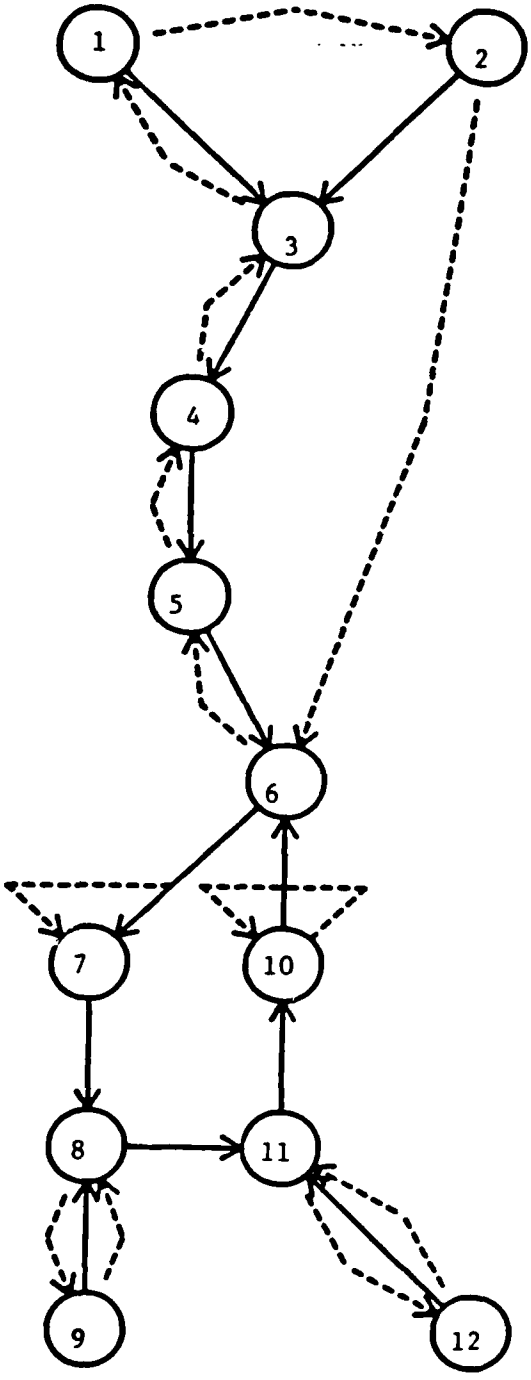


Figure 7-12. Pivot Type 4 (Cont.)



After Entire Routine

Figure 7-12. Pivot Type 4 (Cont.)

7.1.3.5 Pivot Type 5

Pivot type 5 occurs when the entering head and tail are in the same component, but in different trees. The exiting arc occurs on the cycle (otherwise it would be pivot type 2). If the predecessor path from the entering tail node to (and around) the cycle is compared to the path from the entering head, one of theses path will reach EXIT first. Denote the node with the path to first reach EXIT as LEAVE and the other as STAY. Denote the corresponding cycle nodes as CYC_LEAVE and CYC_STAY (see Figure 7-13).

The algorithm for pivot type 5 is:

```
NODE := LEAVE;
PREV := STAY;
while NODE <> CYC_LEAVE do
      ISOLATE(NODE);
      TEMP := PRED(NODE);
      PRED(NODE) := PREV;
      PREV := NODE;
      NODE := TEMP;
endwhile;

PRED_EXIT := PRED(EXIT);
REV_CYC_REROOT(CYC_LEAVE,EXIT);

NODE := STAY;
PREV := LEAVE;
while NODE <> STAY_CYC do
      ISOLATE(NODE);
      PREV := NODE;
      NODE := PRED(NODE);
endwhile;

CYC_REROOT(PRED_EXIT,CYC);
```

See Figure 7-14.

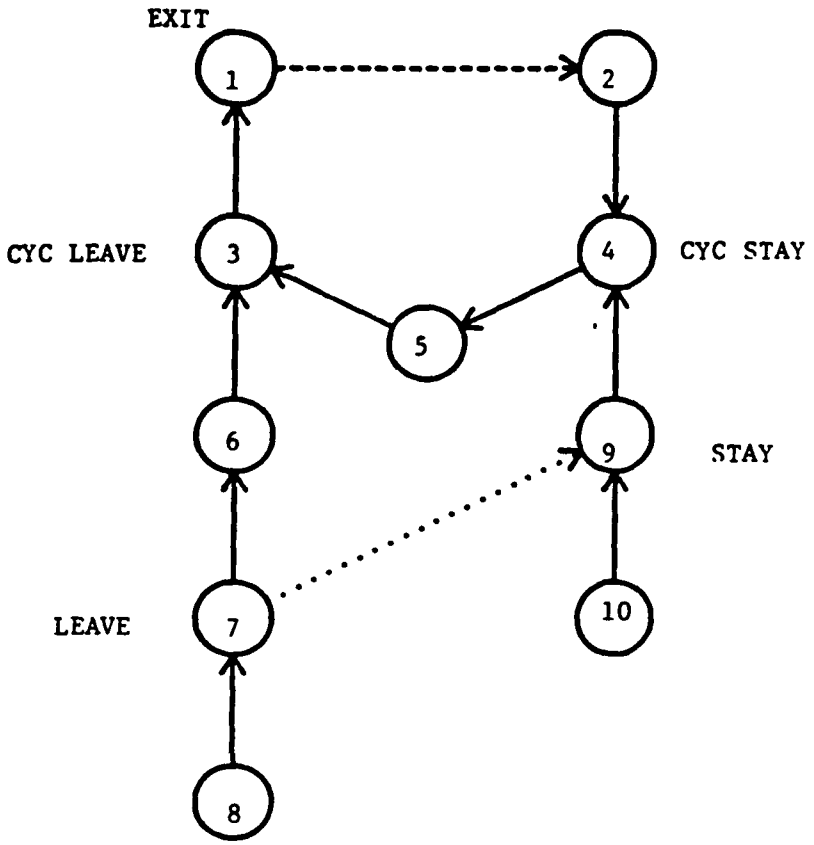
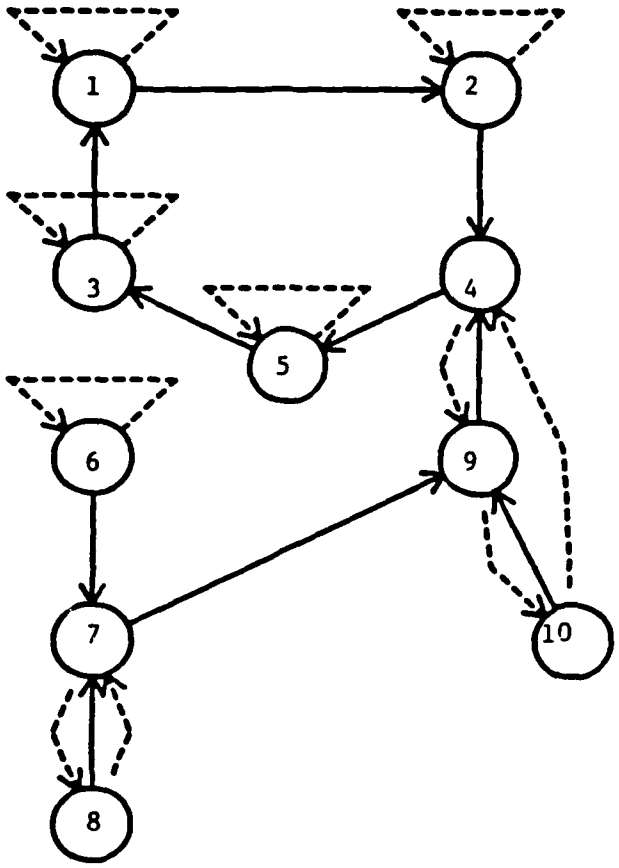
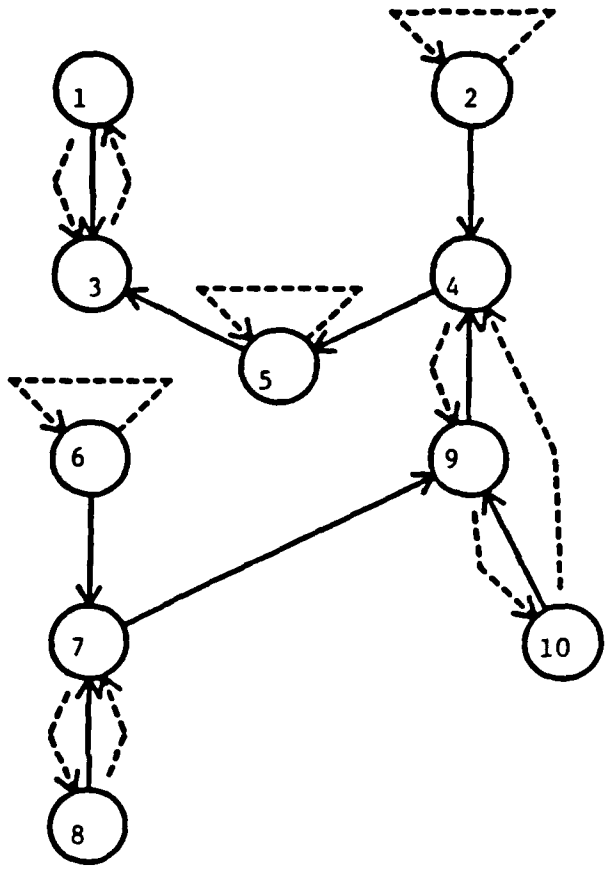


Figure 7-13. Pivot Type 5 - Initial Position

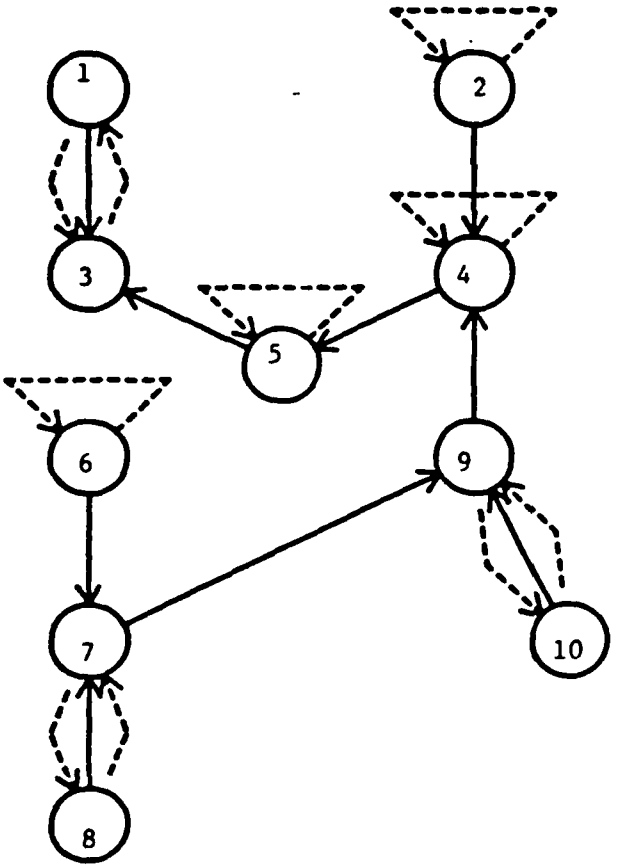


After Section A

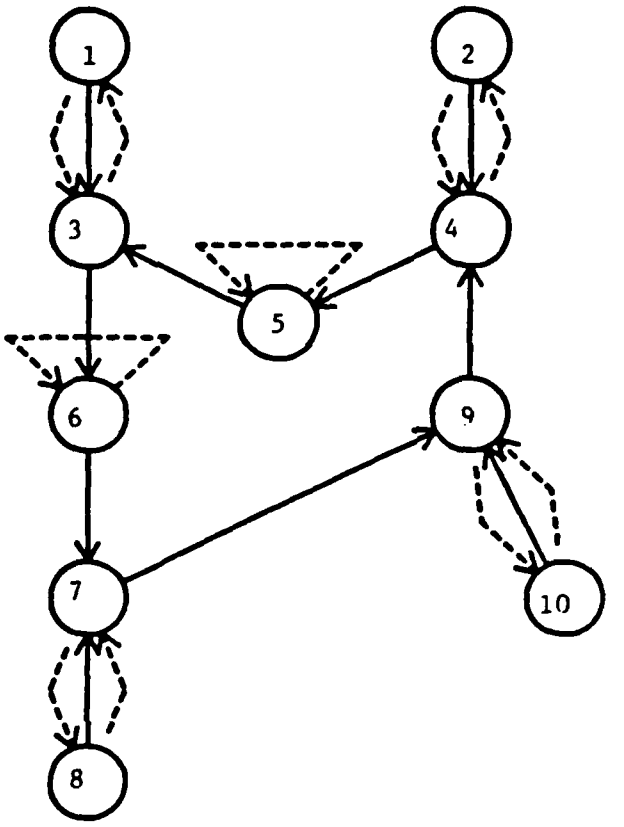


After Rev-Cyc-Reroot (3,1)

Figure 7-14. Pivot Type 5 During Pivot



After Section B



After Pivot

Figure 7-14 Pivot Type 5 During Pivot (cont.)

7.1.3.6 Pivot Type 6

Pivot type 6 occurs when the entering arc has its nodes in different components and the exiting arc occurs on a cycle. (If the exiting arc is not on a cycle then the pivot type is 2).

Denote the node of the entering arc that is in the same component as the exiting arc as LEAVE and the other end as STAY. Let CYC be the cycle node associated with LEAVE. See Figure 7-15.

The algorithm for pivot type 6 is:

```
REROOT(CYC,LEAVE);
PRED_EXIT := PRED(EXIT);
REV_CYC_REROOT(CYC,EXIT);
CYC_REROOT(PRED_EXIT,CYC);
HANG(STAY,LEAVE);
```

See Figure 7-16.

7.2 Updating the Duals

In order to calculate reduced costs quickly, dual values associated with the node are maintained in core. Because the dual variable calculation is a computationally expensive operation, it is fortunate that only a small number of dual values change at each iteration. The duals that change are exactly those whose node receives a new LEVEL value.

The key to calculating duals is that reduced cost for

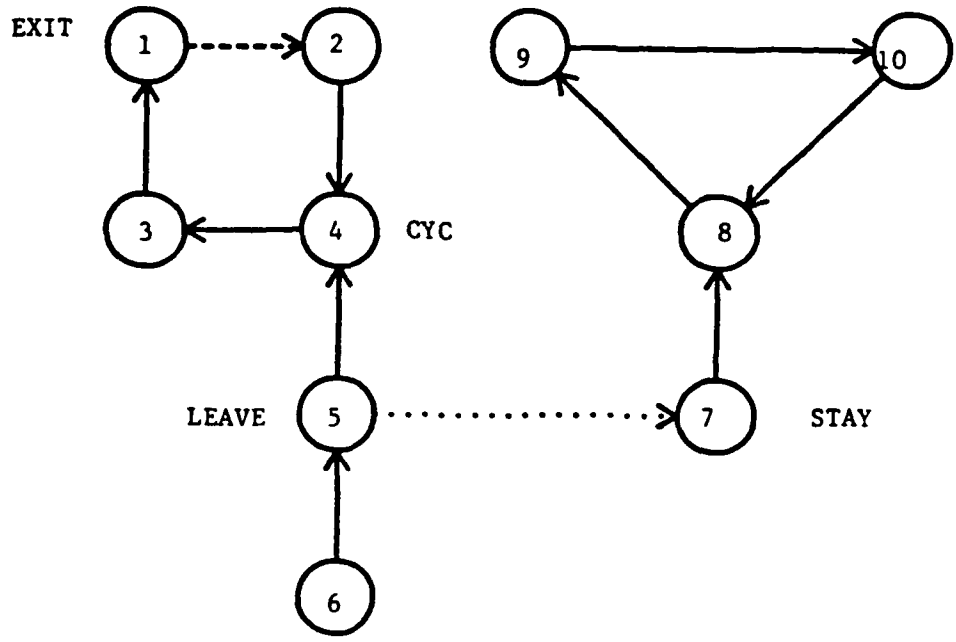
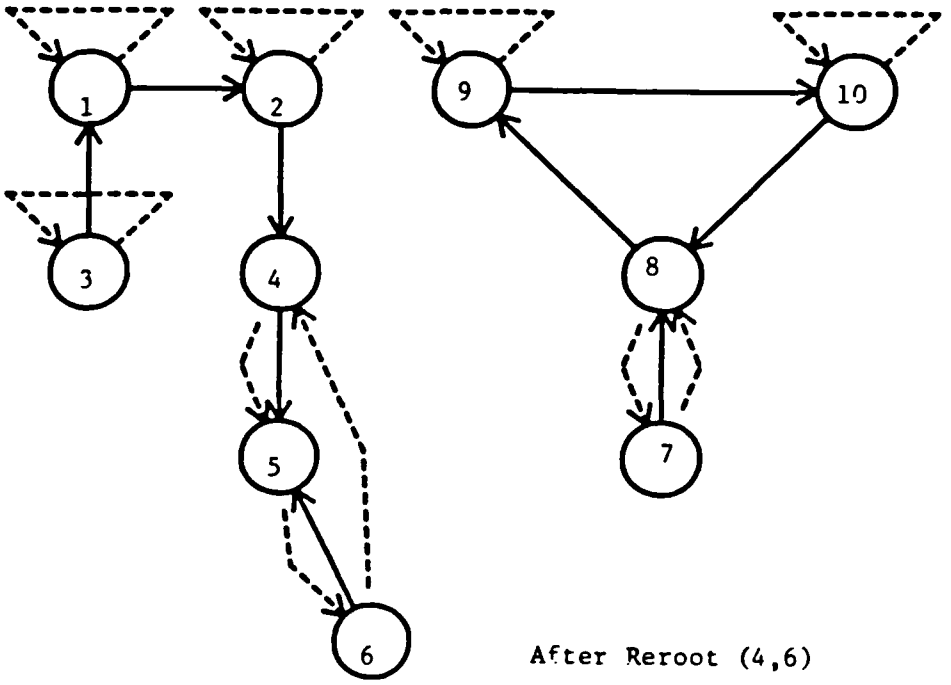
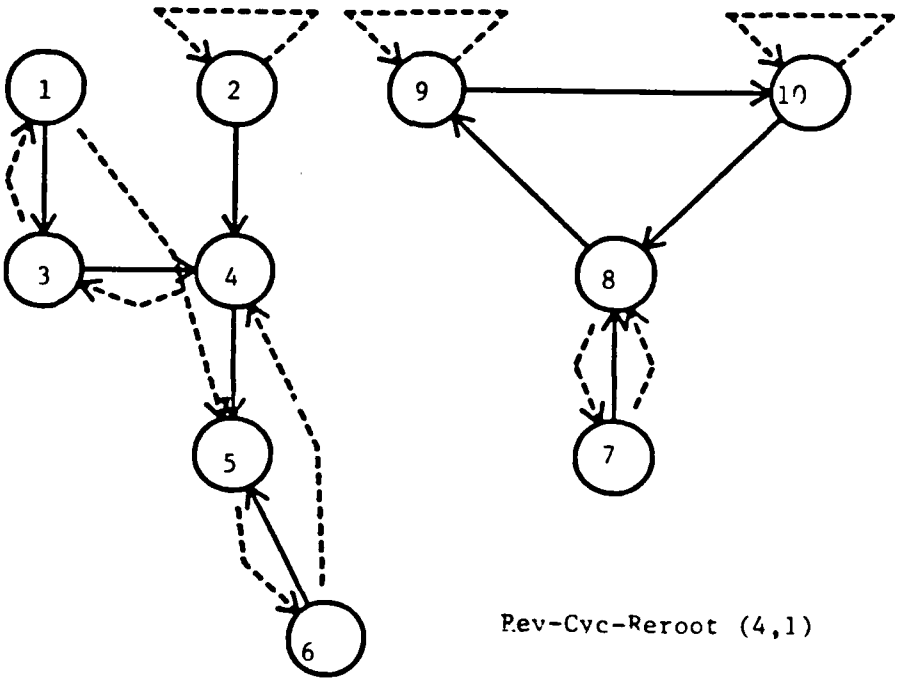


Figure 7-15. Pivot Type 6 - Initial Position

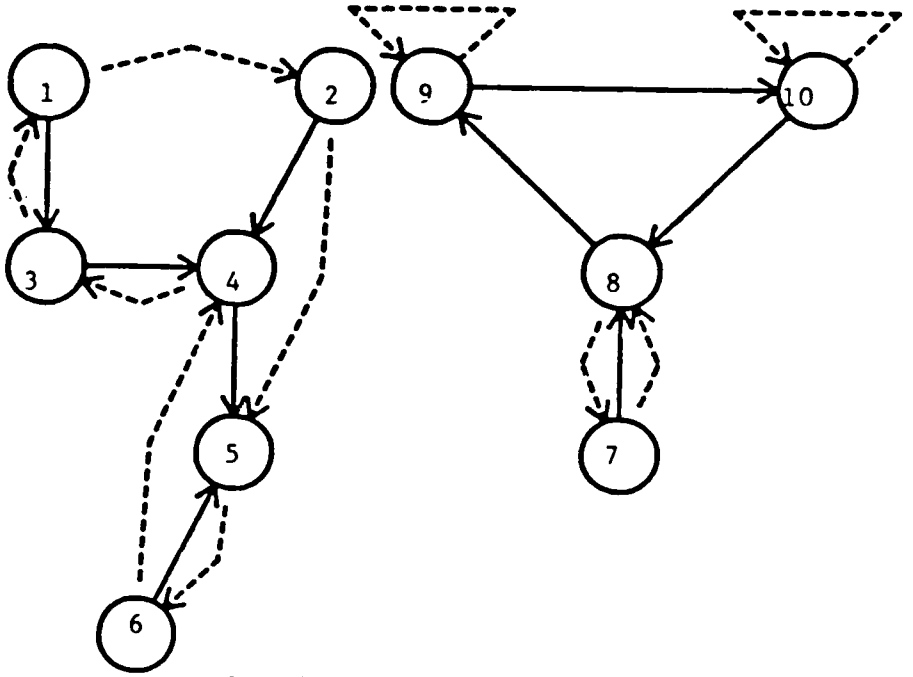


After Reroot (4,6)

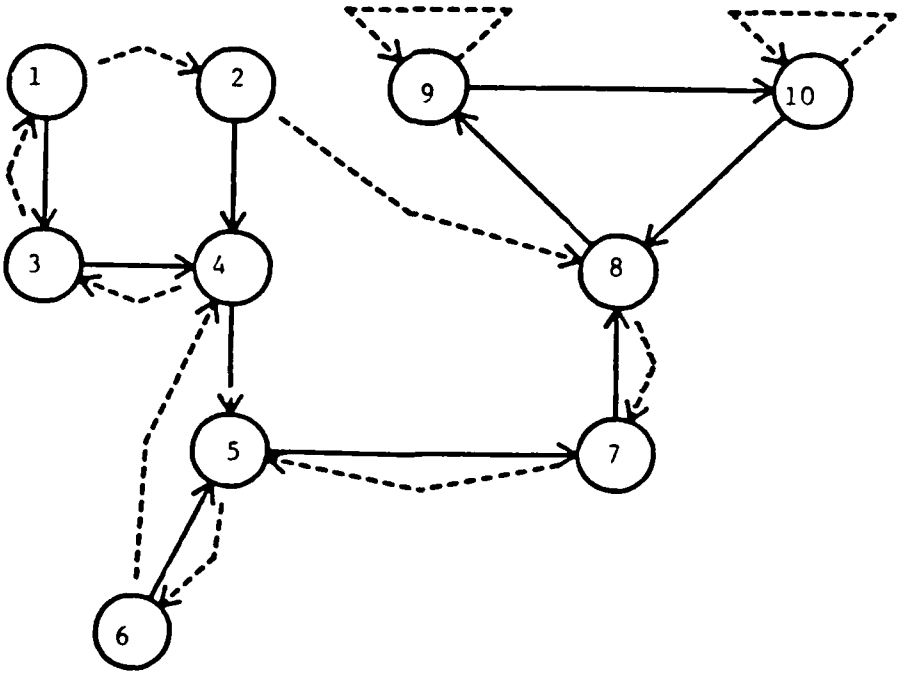


Rev-Cyc-Reroot (4,1)

Figure 7-16. Pivot Type 6 - During Pivot



After Cyc-Reroot (4,2)



After Pivot

Figure 7-16. Pivot Type 6 - During Pivot (Cont.)

arcs in the basis are always zero. The reduced cost calculation is (from Section 5)

MULT * DUAL(HEAD) - DUAL(TAIL) - COST

Duals for generalized networks are uniquely determined. Calculation of duals for nodes on the cycle is involved and is revieved in Section 7.2.1. The dual for a node off the cycle is based solely on the dual value of its predecessor. This is shown in Section 7.2.2.

7.2.1 Dual Values on the Cycle

Given a cycle with k nodes there are k arcs between them. Each arc creates one reduced cost calculation. Therefore, there are k linear equations to find k unknown duals.

If one dual on the cycle is given, the rest can be obtained by traversing the cycle with the PRED values. The method to find one dual on the cycle follows.

Let the nodes on the cycle to be 1 ... k, with PRED(i) = i+1 and PRED(k) = 1. The following calculates DUAL(1):

```
TOT_SUM := 0;
TOT_MULT := 1;
NODE := 1;
repeat
     if ARC(NODE) < 0 then (* ARC is from PRED(NODE) to NODE *)
          TOT_MULT := TOT_MULT / MULT(NODE);
          TOT_SUM := TOT_SUM - TOT_MULT * COST(NODE);
     else
          TOT_SUM := TOT_SUM + TOT_MULT * COST(NODE);
          TOT_MULT := TOT_MULT * MULT(NODE);
     endif;
     NODE := PRED(NODE);
until (NODE = 1);

DUAL(1) := TOT_SUM / (1-CMULT(1));
```

Duals of cycle nodes associated with self-loops are easier. For this case, the dual is simply the cost divided by the self-loop multiplier.

7.2.2 Dual Values not on the Cycle

The dual value for a node off the cycle is calculated as:

```
if ARC(NODE) < 0 then
     DUAL(NODE) := (DUAL(PRED(NODE)) + COST) / MULT;
else
     DUAL(NODE) := DUAL(PRED(NODE)) * MULT - COST;
endif
```

It is important that the dual variable updates are performed in the correct order. For every node, the dual for the PRED of the node must be calculated before the dual of the node can be calculated. It is for this reason that THREAD is defined to be the preorder traversal. Following the thread will ensure update of the PRED of any node before the node itself; the DUAL update can be accomplished at the same time. This reduces the number of times any node must be examined during a pivot, yielding in a significant reduction in computation time.

7.2 Updating the Flows

The final structure to update is the basic flow values. Most of the work in this update was accomplished during the calculation of the exiting arc. The updated column was the change in flow on the basic arcs if one unit of flow was put on the entering arc. This

allowed a calculation of the maximum flow that could be placed on the entering arc without violating any bounds. This value is multiplied by the updated column and added to the current flows to produce the updated flows.

Algorithmically, it is easier to perform this update while the basis is being updated. As each node is visited it is a simple matter to calculate the amount the flow will change and update FLOW accordingly. Some nodes are not visited during the basis update and those nodes must be visited solely to update the flows. For instance, in pivot type 1, no basis update is performed but the flows must be updated. All nodes visited in the calculation of the exiting arc must be revisited to update the flows.

8.0 OTHER CONCERNS

Major portions of the primal simplex method have been covered. Given a basis it is possible to find an arc to enter the basis, determine the arc to exit the basis and update the basis. The only other step of the primal simplex method is to find an initial basis. Section 8.1 gives two methods for doing this.

One other concern is the efficiency of the generalized network primal simplex method when solving a pure network. It is generally thought that a pure network code will execute two or three times faster than a generalized network code. Some of this improvement is due to the fact that generalized networks must keep track of the multipliers while the pure network has one less piece of data for each arc. Some efficiency relates to the structure of the pivots used with pure networks. This pivot and basis structure can be used within a generalized network code, so pure networks and "almost" pure networks can be solved more quickly than generalized networks with the generalized network simplex method. This is detailed in Section 8.2.

8.1 Initial Basis

The starting basis has a large effect on time required by the primal simplex method. Ideally, if the optimal basis is used as the starting basis, then no pivots need be performed. The concept of quality of the starting basis is difficult to quantify, although it seems reasonable to assume that a good starting basis is one which

has a large number of arcs that are also in the optimal basis.

If too much time is spent in trying to find a good starting basis, the total execution time might be larger than using a poorer, but easier to find, basis.

An initial basis must satisfy the following conditions:

1) There must be one arc in the basis for each node.

2) Each component generated by the arcs in the basis must have exactly one cycle, which may be a self-loop.

3) The net flows into and out of each node must be equal to the supply or demand at that node.

4) The flow on any arc not in the basis must be either zero or the capacity of the arc.

5) The flow on any arc must be between zero and the capacity of the arc inclusive.

Section 8.1.1 describes the easiest basis to find: the artificial start basis. By spending essentially no time in creating the initial basis, this basis makes no attempt to guess which arcs will be in the optimal basis.

Section 8.1.2 discusses basis creation methods that try to guess which arcs are optimal. These advanced start methods are often problem specific, although some general purpose methods are possible.

8.1.1 Artificial Start

Associated with each node in a generalized network is a self-loop, representing the slack or surplus variable at the node. If the supply or demand at the node must be satisfied then there is a large cost on the self-loop, otherwise the cost is zero. The artificial

start basis is as follows:

1) The flow on every self-loop is the supply or demand at the node.

2) The flow on every other arc is zero.

The arcs in (1) above form the basis. This basis satisfies the five conditions given in Section 8.1. It is also easy to create. Furthermore the PRED, THREAD, LEVEL and RTHREAD are very easy to calculate. Figure 8-1 illustrates this for the network in Figure 2-3.

The disadvantage of this method is that no attempt is made to determine which arcs will likely occur in the optimal basis. The number of pivots required after an artificial start is probably more than the number required after other types of starts.

8.1.2 Advanced Start

A basis that attempts to contain arcs that will occur in the optimal basis is referred to as an advanced start. Advanced starts can often be determined from the structure of the particular problem types that are being solved. Many, although not all, advanced start strategies adopt the following form:

1) Sort the arcs in decreasing order of attractiveness (likelihood to be in optimal basis)
2) Using the most attractive unexamined arc, place the maximum amount of flow that will keep feasibility at the nodes and the arc on the arc.
3) Repeat Step 2 until all of the arcs have been examined.

Algorithms that adopt the above strategy and place all unallocated supply and demand on the self-loops create a basis that

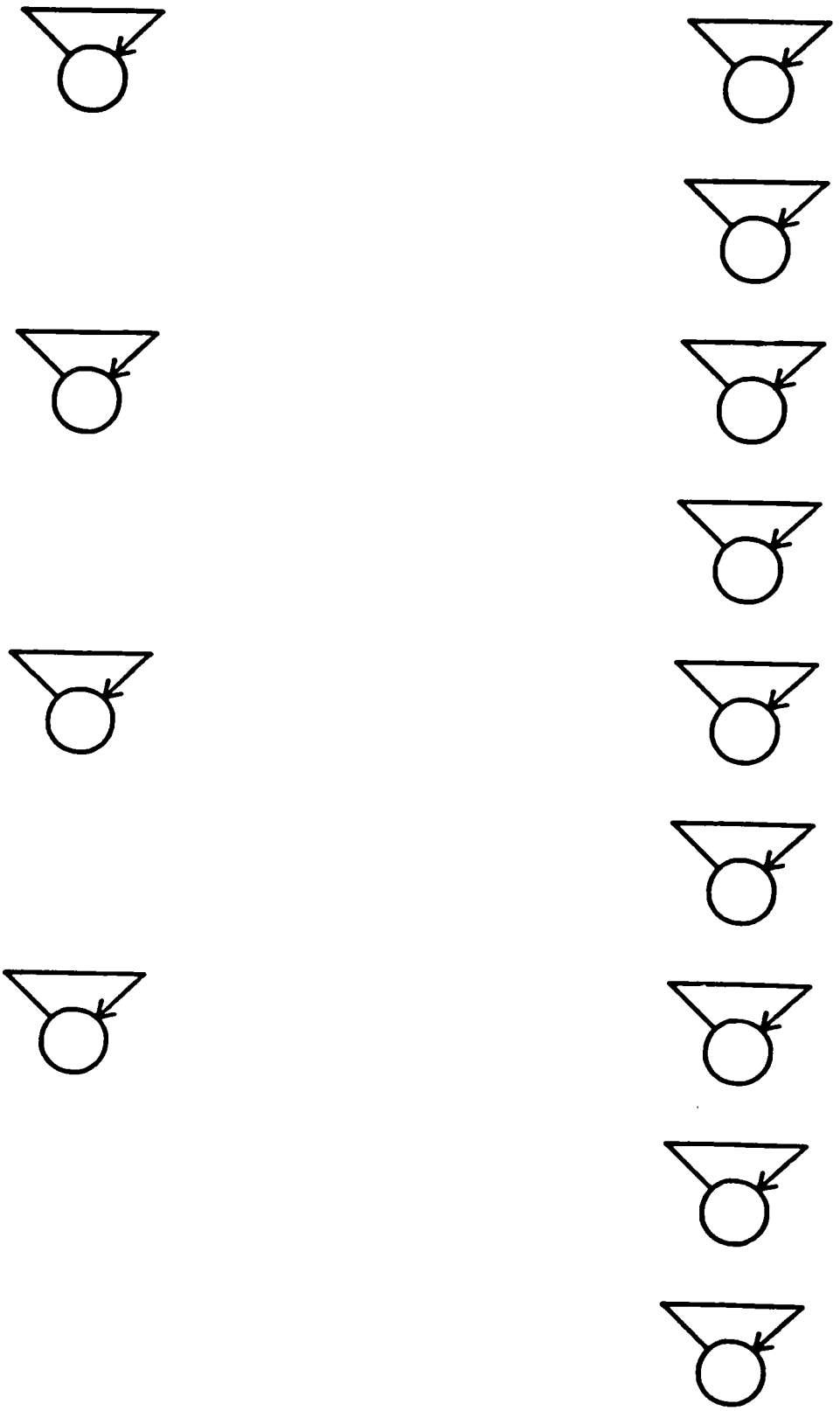


Figure 8-1. Artificial Basis

satisfy the five conditions in Section 8.1 (possibly some self-loops with zero flow will have to be added to the basis). If easy methods of calculating attractiveness are available and if attractiveness is a true measure of the likelihood of being in the optimal basis then advanced starts of this form are generally fairly effective.

In Figure 8-2 the arcs in Figure 2-3 are ordered by increasing cost and an advanced start basis is formed.

8.1.3 Specialization for MRMATE

Arcs of the MRMATE model are already sorted into three classes based on attractiveness. This permits an easy advanced start. Arcs in the zero cost arc file are placed into the basis first. Next, low cost arcs are used. Because there are a large number of high cost arcs, it seems better to ignore them in creating an advanced start and let the rest of the simplex algorithm choose which high cost arcs to use by pivotting them into the basis as needed.

8.2 Effect of Pure Network Structure

Pure networks can be solved more quickly than generalized networks. This is for two reasons:

1) Pure networks have multipliers of one. Hence, rather than using multiplications and divisions, the various simplex calculations can assume the MULT is one.

2) The basis structure of pure networks is simpler than that of generalized networks. Each component of a pure network basis must be rooted at a self-loop.

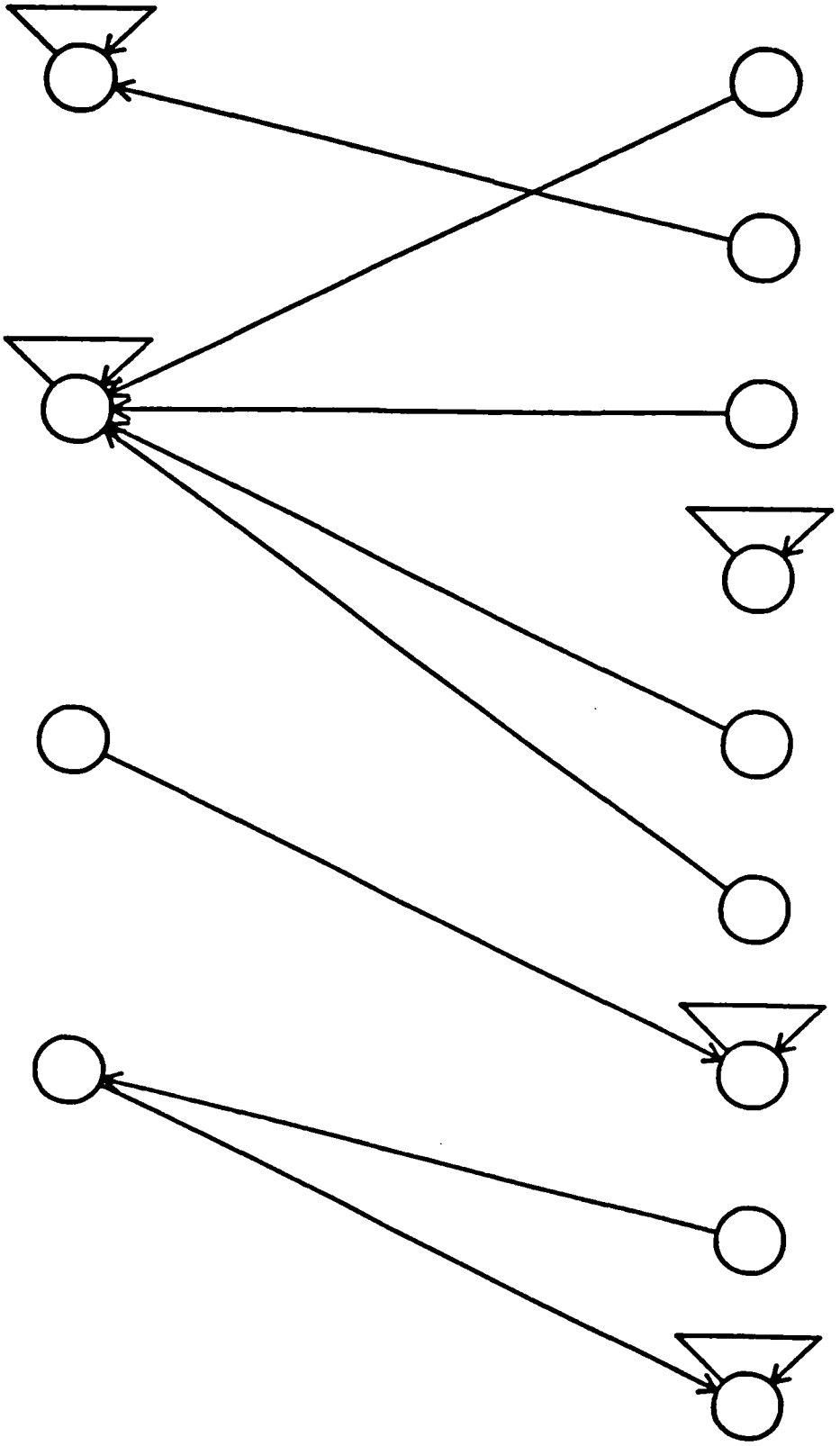


Figure 8-2. Advanced Start Basis

It is possible for a generalized network code to solve pure networks almost as quickly a pure network code by performing the following:

1) A multiplier is compared to 1 before a division or multiplication. If it equals 1 then the division or multiplication is not needed.

2) Data structures are used that are equivalent to pure network data structures in the case of a self-loop root.

Unfortunately, handling the multipliers as suggested in point one will slow down execution for all networks. This decrease in speed may not be excessive but may not be worthwhile if the number of pure arcs is small. Speed increases will occur, however, even in generalized networks, as long as the network has a large proportion of pure arcs.

Data structures used in pure networks are exactly those outlined in Section 4 in the case where the root cycle is a self-loop. Advantages of the basis structure can occur in any generalized network that can be converted to a pure network by scaling the rows and columns of the constraint matrix. The basis advantages can be gained even if the scaling is not performed. An example of a generalized network that can be scaled to a pure network is given in Figure 8-3.

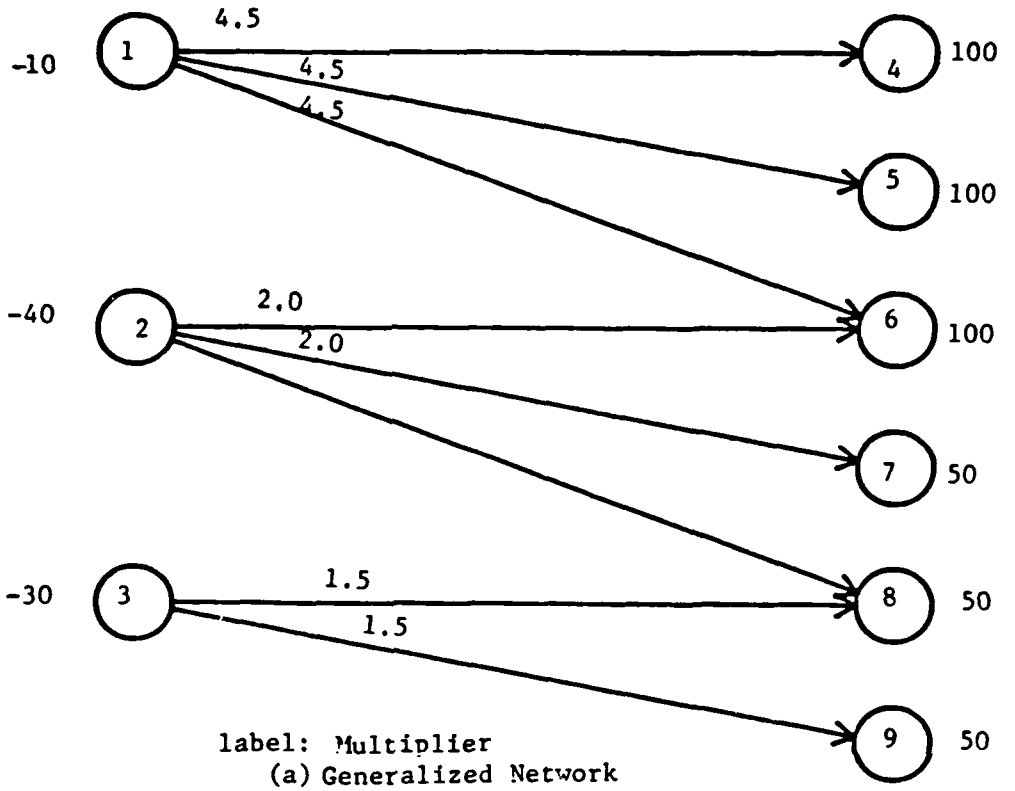
The main advantage of the pure network structure is that all pivots will be of types 1 or 2 (if the self-loops are replaced with artificial arcs and an artificial node, see [3]). No cycles are ever created, so no cycle multipliers need be calculated (Section 4). Calculation of the cycle multiplier is a very expensive operation, since it involves many multiplications and divisions.

It seems likely that a network that is "almost" transformable to a pure network would have many of the same advantages as a network transformable to a pure network. "Almost" must be defined very carefully. For instance, Figure 8-4 shows a network with just one arc with a multiplier not equal to one. However, this network is equivalent to the network in Figure 2-3 and the two networks are solved in almost the same way by a generalized network simplex method. The number of cycle calculations is the same, as is the time required. The reason for this is that non-pure arcs occur in the basis disproportionately for their quantity. This shows the disadvantage of defining "almost" pure networks in terms of the percentage pure arcs.

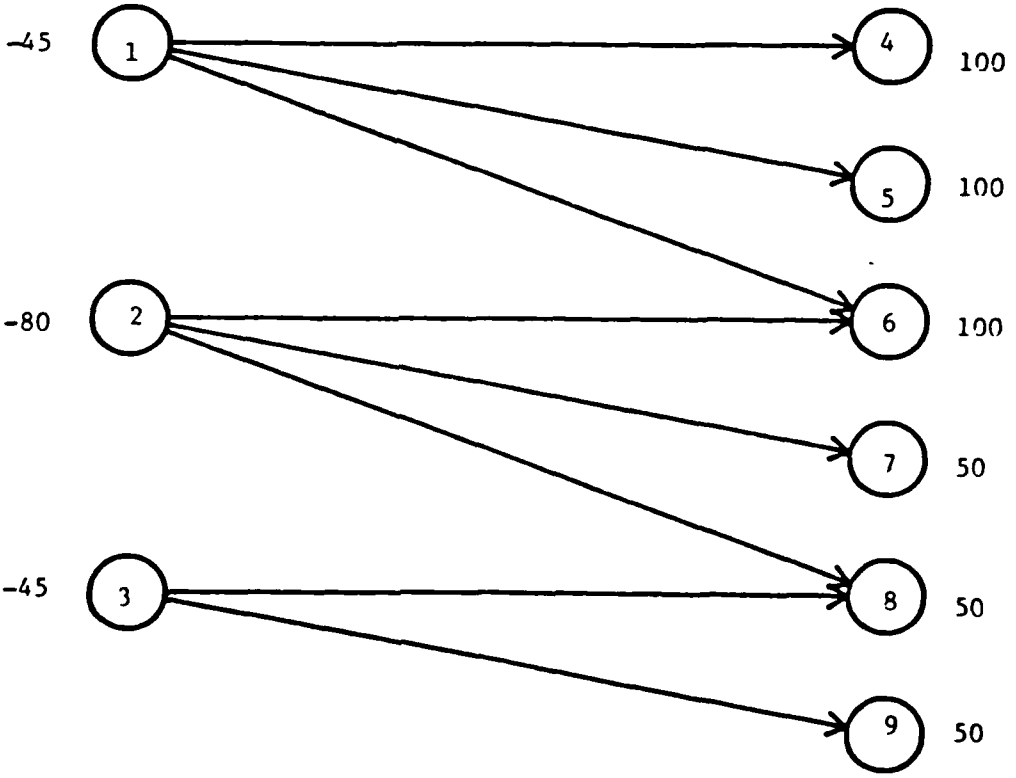
"Almost" pure networks are those whose valid basis tend to have self-loops, rather than cycles. Cycles are formed when the exiting arc occurs on the common part of the backpaths formed by the entering arc. "Almost" pure networks require that for a "typical" basis and entering arc, the exiting arc occurs on the separate parts of the backpath (see Section 6.1).

8.2.1 Specialization for MRMATE

As might be expected, the MRMATE model can be a pure, transformable to pure, or "almost" transformable to pure network. If there are no sea channels then all of the multipliers in the network are one, so a pure network results. If there are no air channels then the network is transformable to a pure network. This is equivalent to using the volume of the movement requirement rather than the weight.



label: Multiplier
(a) Generalized Network



all multipliers are +1 (b) Equivalent Pure Network

Figure 8-3. Generalized Network Transformable to Pure Network

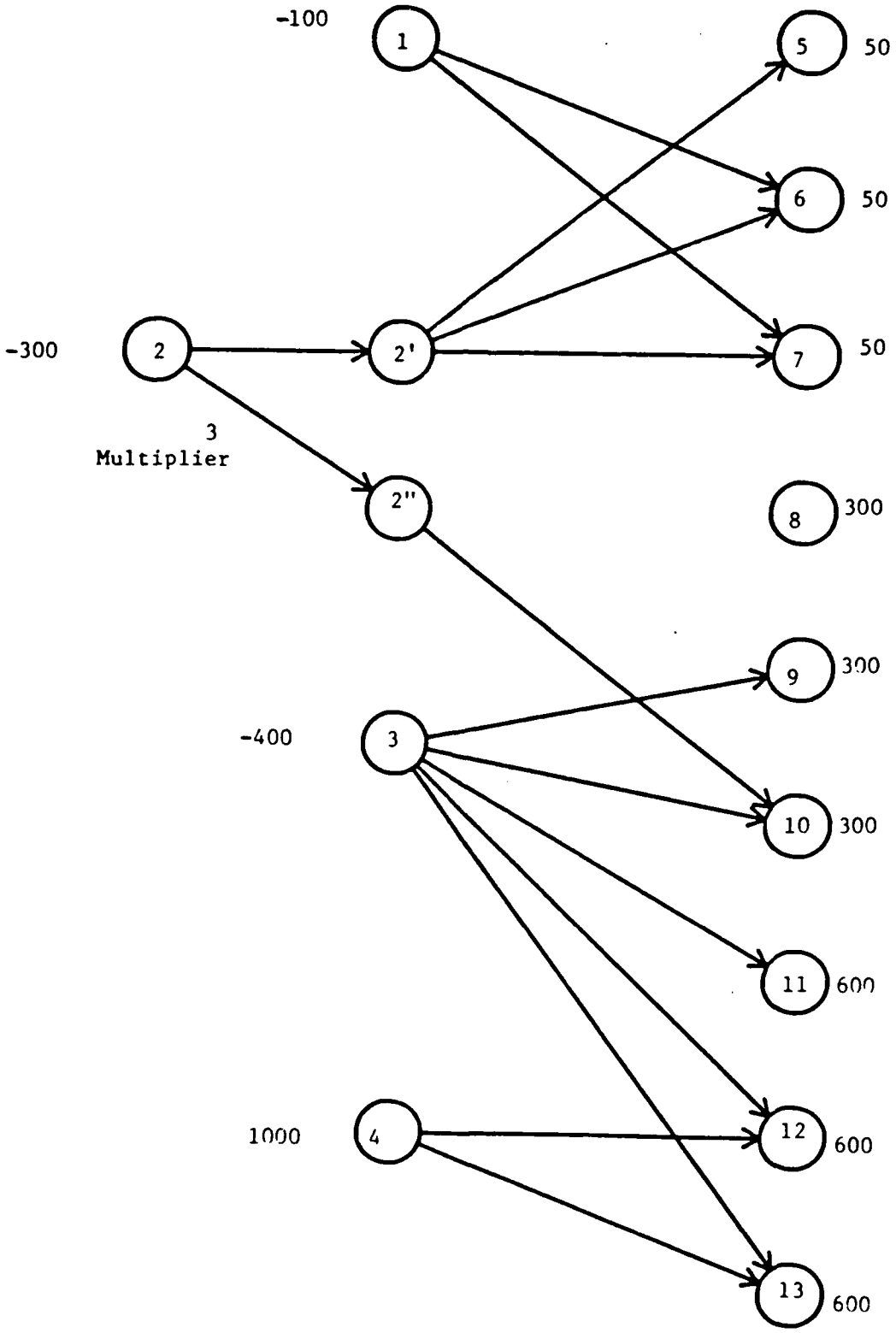


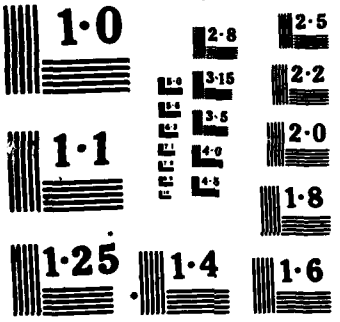
Figure 8-4. Equivalent "Almost Pure" Network

If there are far more air channels than sea channels, or more sea channels than air channels, then the resulting network is "almost" transformable to a pure network as long as the less numerous channels are not disproportionately important. This is due to the following. Given a network with far more air channels than sea channels, a basis and an entering arc, if the channels are equally important, the typical entering arc will be an air channel arc. The backpaths represent the channels that will have to change flows in order to permit the entering arc to join the basis. Typically, this will only involve air channels. This means that the exiting arc cannot occur on the common part of the backpath, since a cycle with a cycle multiplier of 1 would be created. A similar argument holds for more sea channels than air.

How many more is "far more"? Certainly 75 air channels and 25 sea channels will not exhibit much pure network structure, since most backpaths will contain both air and sea channels. Also, 99 sea channels and 1 air channel will exhibit a lot of pure network structure, since most pivots will reassign flow among the sea channels. The effect of pure network structure is further examined in [4].

AD-A168 298 GENERALIZED NETWORK IMPLEMENTATIONS(U) GEORGIA INST OF TECH ATLANTA PRODUCTION AND DISTRIBUTION RESEARCH CENTER J J JARVIS ET AL. 1986 PDRC-86-03 2/2

UNCLASSIFIED N00014-85-C-0797 F/G 5/1 NL



REFERENCES

[1] Bazaraa, M.S. and J.J. Jarvis, Linear Programming and Network Flows, John Wiley and Sons, New York, 1977.

[2] Brown, G.G. and R. McBride,"Solving Generalized Networks", Management Science, 30, 12, pp 1497-1523.

[3] Jarvis, J.J., H.D. Ratliff, D.E. Eisenstein, A.V. Iyer, W.G. Nulty, and M.A. Trick, "System Description: SYSTEM FOR CLOSURE OPTIMIZATION AND PLANNING EVALUATION (SCOPE)", PDRC Report 84-09, Georgia Institute of Technology, 1985.

[4] Jarvis, J.J., H.D. Ratliff and M.A. Trick, "Generalized Network Results on a Microcomputer", PDRC Report (to be published), Georgia Institute of Technology, 1986.

[5] Kennington, J.L. and R.V. Helgason, Algorithms for Network Programming, John Wiley and Sons, New York, 1980.

END

DTIC

7-86